Verification and Validation of Security Protocol Implementations

Nicholas O'Shea

Doctor of Philosophy

Laboratory for Foundations of Computer Science

School of Informatics

University of Edinburgh

2010

Abstract

Security protocols are important and widely used because they enable secure communication to take place over insecure networks. Over the years numerous formal methods have been developed to assist protocol designers by analysing models of these protocols to determine their security properties. Beyond the design stage however, developers rarely employ formal methods when implementing security protocols. This may result in implementation flaws often leading to security breaches.

This dissertation contributes to the study of security protocol analysis by advancing the emerging field of implementation analysis. Two tools are presented which together translate between Java and the LySa process calculus. Elyjah translates Java implementations into formal models in LySa. In contrast, Hajyle generates Java implementations from LySa models. These tools and the accompanying LySa verification tool perform rapid static analysis and have been integrated into the Eclipse Development Environment. The speed of the static analysis allows these tools to be used at compile-time without disrupting a developer's workflow. This allows us to position this work in the domain of practical software tools supporting working developers.

As many of these developers may be unfamiliar with modelling security protocols a suite of tools for the LySa process calculus is also provided. These tools are designed to make LySa models easier to understand and manipulate. Additional tools are provided for performance modelling of security protocols. These allow both the designer and the implementor to predict and analyse the overall time taken for a protocol run to complete.

Elyjah was among the very first tools to provide a method of translating between implementation and formal model, and the first to use either Java for the implementation language or LySa for the modelling language. To the best of our knowledge, the combination of Elyjah and Hajyle represents the first and so far only system which provides translation from both code to model and back again.

Acknowledgements

I am sincerely grateful to Stephen Gilmore for his endless guidance and support over the last few years, without which this work would be considerably poorer and my time considerably less pleasant. Thanks to Ian Stark for acting as second supervisor and in particular for his comments on the work in Chapter 7. I thank the Engineering and Physical Sciences Research Council for funding this work.

Many thanks to Han Gao, Ender Yuksel, Jose Quaresma and all of the students in the Language Based Security course at DTU who provided feedback on the LySa Toolkit in Eclipse. Equally thanks are owed to all those in Lyngby and Pisa who have contributed to LySa and without whom this work would not be possible. Thanks should also go to all the anonymous reviewers whose comments were often helpful, sometimes contradictory but always appreciated. I would like to express my gratitude to Hanne Nielson and David Aspinall for acting as my external and internal examiner respectively. Thanks to them the viva experience was both enjoyable and constructive and their comments helped to clarify this work.

Special thanks to all of my colleagues in the School of Informatics especially my various office mates over the years who have consistently helped, amused and encouraged me along this journey. Thanks to my friends who I am reasonably certain will never read a word of this work but to whom I am grateful for their friendship and ability to distract me from my work. Finally thank you to my family, especially my parents, for their unwavering love, support and inspiration.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualication except as specified. Preliminary work that later lead to Chapter 5 was my final year project as an undergraduate. Some of this work has previously been published in [58, 59, 60].

(Nicholas OShea)

UIJT QBHF JOUF OUJP OBMM ZMFG UCMB OL

Table of Contents

Chapter 1

Prolegomenon to a Thesis

The study of security protocols has never been more important. Typically such study has focused on analysing formal specifications of security protocols ever since [22] stated that these protocols "are the basis of security in many distributed systems, and it is therefore essential to ensure that these protocols function correctly." Such forms of analysis have evolved since then and the importance of security analysis has only grown, however it is apparent that this is only one piece of the puzzle to "ensure that these protocols function correctly." Beyond the security properties of a protocol, other properties such as liveness and timely completion are also important.

Although formal methods continue to be used and developed within academia, they are underused by the software development community as a whole. While specifications of security protocols are meticulously studied for any potential attacks, flaws can emerge at the implementation stage. This may be due to developers working from incorrect specifications or because of simple programming errors. Specifications often leave out crucial information that a formal model or code fragment would not, and trust in the developers to correctly implement their intentions. Many developers do not have sufficient experience to be able to spot any errors they make as flaws in security protocols are often extremely subtle. Thus we can see that some form of analysis of the implementation is necessary. While there are general-purpose static analysis tools that focus on applying formal methods to source code[33, 42], they typically help a developer to create an application that will compile but does not necessarily correctly implement the developer's ideas. The separation of these two tasks is commonly referred to as validation and verification. Informally, validation is said to be "building the right program" while verification is "building the program right". Creating automated tools for validation is hard. By focusing on a specific application domain where there

are formal methods that can be applied, the validation problem becomes more akin to verification. Here we focus on analysing security properties of security protocols.

The two options available going forward are automatically generating code based on an analysed formal model or extracting a formal model from an implementation.

From Model to Implementation

Taking a formal model of a security protocol and converting it into a working implementation is the easier direction to start with. As the model language is more specialised than the implementation language it is easier to divine the intention of a protocol. The downside is that once a developer takes the output of the translation they cannot make any changes and remain certain that they are still validly representing the model.

From Implementation to Model

It is much harder to determine the purpose of implementation code as the scope of the language is so great. There are multiple methods of achieving the same goal with very different syntax. However, it remains the most useful direction as analysing the implementation code allows us to be certain of the security properties of the implementation.

In this thesis we will explore methods of bridging the gap between formal models and protocol implementations. We will present a unique two-way translation between LySa and Java allowing translation from model to implementation and vice-versa. We are focusing on security protocols which use cryptography to achieve security goals such as authentication or confidential message sharing. As such, throughout this work we interchangeably use the terms “security protocol”, “cryptographic protocol” and “authentication protocol” to refer to this family of protocols.

1.1 Contribution Overview

The aim of this thesis is to illustrate that:

“Analysing implementations is as important as analysing models and allowing developers not versed in the details of formal methods to use these forms of analysis improves their value.”

The specific contributions towards these aims are as follows:

- A Java API providing necessary features for implementing cryptographic protocols.
- A tool which translates from an implementation using this API to a formal model

in the LySa process calculus. This tool is named Elyjah.

- An accompanying tool named Hajyle which generates Java implementations from a LySa model.
- A sandbox environment for LySa called LyTE which works in the Eclipse IDE[31].
- Tools for performance analysis of cryptographic protocols using the PAM framework.

Figure 1.1 shows how all of these tools work together providing a cohesive suite of tools for the generation and analysis of security protocols. The LySatool is a pre-existing tool, full details of which can be found in [21].

Elyjah was among the very first tools to provide a method of translating between implementation and formal model, and the first to use either Java for the source language or LySa for the target language for the translation. To the best of our knowledge, the combination of Elyjah and Hajyle represents the first and so far only system which provides translation from both implementation to model and back again.

The LySa Toolkit in Eclipse has been used by students being taught LySa as part of the Language Based Security class at the Technical University of Denmark.

1.2 Thesis Development

This work started with development on Elyjah. This name is derived from the first and last letters of Edinburgh, where this work was undertaken, and the opening two letters of both the source and target languages, LySa and Java. As work continued on Elyjah it became clear it would be useful to have a method to quickly check that the generated LySa is both valid and accurate. This directly led to the development of an editor for LySa and a tool for summarising LySa models as protocol narrations. When the benefit of these tools became clear it was decided to extend the functionality. Identifying teaching as a potential target use for the under development tool suite, a tool for generating visual representations of LySa models was added. After presenting LyTE at a conference it came to the attention of those teaching LySa at DTU who used it on their course. While giving a guest lecture on this course, further ideas for improvement were suggested, many of which were implemented in updates which were used on the course the following year. When writing this dissertation, the LyTeX tool was

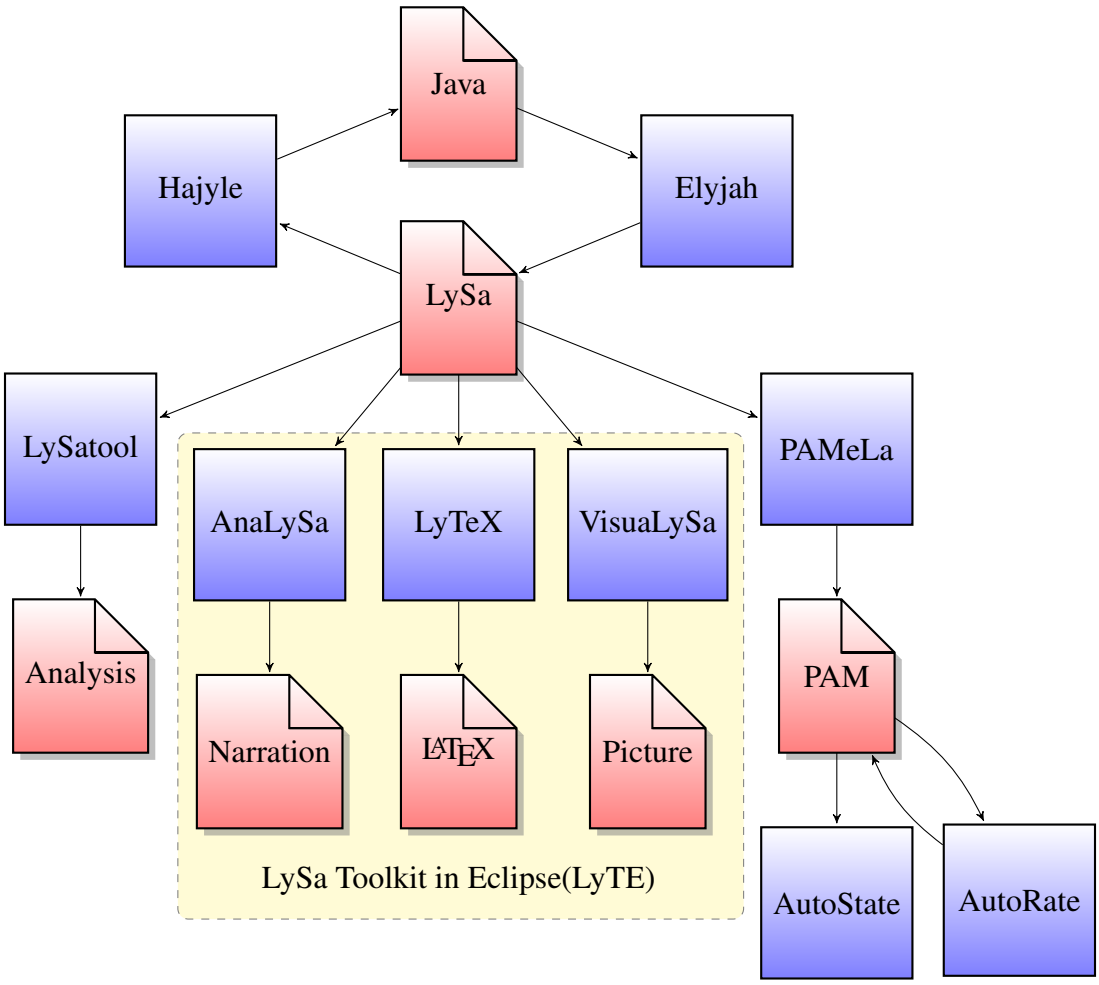


Figure 1.1: Connectivity of Thesis

developed to allow consistent LaTeX representations of LySa models in a convenient manner.

During the development of LyTE a LySa parser was created which motivated the development of additional LySa analysis tools that did not fit in the more general scope of LyTE. As enquiries had already been made at conferences about the possibility of model to code translation, and as an informal understanding of the relationship between the languages was understood from development of Elyjah this challenge was tackled first. The resulting tool was named Hajyle as it performs the reverse translation to Elyjah. The LySa parser was also used to provide performance analysis of cryptographic protocols. This was motivated by the development of the PAM framework allowing sophisticated analysis without creating bespoke tools.

Finally, the work presented in the formal chapter was initially motivated by com-

ments received from reviewers and fellow participants from conferences where Elyjah was presented. While undertaking the work however, it proved useful to provide an alternative viewpoint on the work, which helped to check for bugs in Elyjah and Hajyle. As such, it is a recommended step for those who are working on similar translation tools and should be undertaken in parallel with tool development as both avenues inform the other.

1.3 Outline of the Thesis

Chaper 2 provides an introduction to cryptographic primitives and the protocols that are based around them. The LySa language is introduced along with basic examples. Methods of analysing formal models, in particular LySa representations, of these protocols are introduced.

Chapter 3 presents a toolkit for the LySa language which has been named LyTE. This toolkit was originally designed to help verify that the output of the Elyjah tool in Chapter 5 was valid. After proving useful for this purpose and based on suggestions from users it was expanded to provide useful functionality for those exploring LySa having come from Elyjah or learning LySa by any other means.

This work was previously presented in [58] and for the past two years has been used by the Language Based Security course in DTU.

Chapter 4 introduces a method for analysing the performance of cryptographic protocol models in the LySa language. An important part of security is accessibility to information thus making it imperative that the protocol completes within a reasonable time.

This work was previously presented in [60].

Chapter 5 presents the Java-LySa API and the Elyjah tool. The API provides a structured framework for implementing cryptographic protocols in the Java programming language. The Elyjah tool converts such implementations into LySa so that they can be analysed and the security properties of the implementation revealed.

This work was previously presented in [59] and preliminary work on the topic earned me the ScotlandIS Young Software Engineer of the Year Award in 2006.

Chapter 6 introduces the Hajyle tool. As implied by the name, this tool performs the opposite translation to Elyjah and generates Java implementations using the Java-LySa API from LySa models.

Chapter 7 presents a formalisation of the framework used to implement cryptographic protocols and discusses the equivalence between JaLAPI implementations and models expressed in LySa.

Chapter 8 concludes this dissertation by discussing both the work presented herein and potential future work that has arisen as a result.

1.4 Prior Knowledge and Chapter Dependencies

While we have endeavoured to keep the material accessible, prior experience with an object-based programming language such as Java is certainly advisable and knowledge of process calculi of the π variety will help with understanding LySa examples. Although the material covered in Chapter 2 is useful reading for all subsequent chapters, Chapters 3 and 4 are independent of both each other and the material in Chapters 5, 6 and 7 which are closely related and should be read in order.

Chapter 2

Cryptographic Protocols and the Analysis Thereof

This chapter presents the background material for the thesis. I will present a brief introduction to cryptography and expand into details on cryptographic protocols and the analysis thereof. In Section 2.5 the LySa process calculus will be introduced, defined and explained. The following work builds upon this. The motivation for examining implementations of protocols is presented in Section 2.8. This section gives an outline of the objectives of the work in later chapters.

2.1 Cryptography

The science of encrypting information and the opposing struggle to break another's system (Cryptanalysis) has been at the heart of many conflicts both political and military, specific examples such as the cracking of the Nazi's Enigma machine codes contributed to the end of the Second World War. Modern cryptography has its roots in the mid 1970s when a draft of the Data Encryption Standard was published seeking feedback from industry experts. Another key component of cryptography was introduced by Whitfield Diffie and Martin Hellman called asymmetric key encryption[29]. The publication of these advances saw an uptake in public interest and academic development of cryptography. Although such widespread publication of techniques designed for confidential communication may appear counter-intuitive such openness works to minimise the flaws in these works. Publication of cryptographic advances fits with Kerckhoff's principle stated in 1883. This idea has been widely adopted in security circles and states that "There is no secrecy in the algorithm, it's all in the key". Sim-

Figure 2.1: Basic Cryptographic Scenario

ply put this means that a system should be secure even if the enemy has access to the algorithm's code and knows the workings of the system.

Converting a plaintext message into one that an eavesdropper cannot understand is known as *encryption*. The result of this encryption depends on the algorithm used, the plaintext message and a further parameter referred to as a *key*. We represent an encrypted message with plaintext M and a given key, K, as $\{M\}_K$. Turning this unreadable message into the original text is referred to as *decryption*. This process requires the correct key in order to decrypt the message correctly. Two differing subsets of cryptography are described below.

2.1.1 Symmetric Cryptography

Symmetric encryption refers to a method of encrypting data where the same key is used for both the encryption and decryption processes. In this thesis we will be following the tradition of using characters Alice and Bob to demonstrate cryptographic scenarios. In Figure 2.1 we see a typical scenario, Alice and Bob want to communicate confidentially with each other in the presence of a third-party who can intercept their communication. If they already share a cryptographic key, then Alice can encrypt a message with this key, transmit the encrypted message to Bob who can then decrypt the message using his key. As long as the third-party attacker does not know the key and a suitably complicated encryption algorithm is used, this communication is secure.

2.1.2 Asymmetric Cryptography

Asymmetric cryptography differs from symmetric in that different keys are used to encrypt and decrypt an encrypted message. This provides advantages for authentication and key sharing. A user will generate a pair of keys, one public and the other private. It is not possible for an attacker to derive one key if they know the other. The public

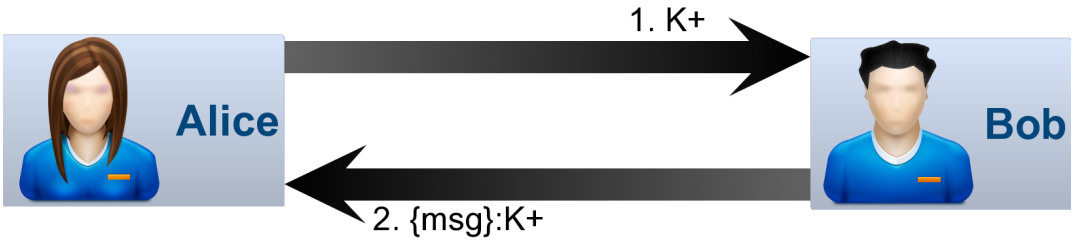


Figure 2.2: Basic Asymmetric Communication

key can be widely distributed while the private key is kept secret. Another user can then encrypt a message with the public key confident that the message can only be decrypted by the accompanying private key which only the intended recipient knows.

A common analogy used to clarify the difference between these two kinds of encryption is that of Alice and Bob sending padlocked boxes to each other. Under symmetric encryption, Alice and Bob both have keys to the same padlock. This padlock is used by Alice to lock a box, into which she has already placed some secret information. This box is then sent to Bob, who uses his key to unlock the padlock and read Alice's message. In asymmetric encryption, when Alice wants to send a message to Bob, she must first request Bob's padlock. She uses this padlock to lock a message inside a box, and send the box to Bob who can then use his key to unlock his own padlock and read the message.

In Figure 2.2 we see a basic asymmetric protocol. Alice sends Bob her public key so that Bob can then encrypt a message to Alice with the intention that this information is secure and authenticated. As we will see in Section 2.2 this protocol has multiple flaws which mean neither of these goals are achieved.

2.1.3 Cryptographic Hash Function

Related to cryptography is the notion of a hash function. This can be thought of as a one-way encryption such that knowing the resulting hash value, it is not possible to calculate the original plaintext. More formally, given a plaintext message M and a hash function H, it should not be possible to calculate M' such that $H(M) = H(M')$. Equally two different plaintext inputs should return different hash values, namely given different $M1$ and $M2$, $H(M1) \neq H(M2)$. Hash functions can be used to prove that a principal knows a secret without revealing the secret or by providing a hash of a message along with the message, to prove the message has not been tampered with.

2.1.4 Specific Algorithms and Cryptanalysis

Work continues on developing more robust and secure algorithms for encryption and hashing. The National Institute of Standards and Technology is currently halfway through a competition [51] to choose a successor to the SHA-1 and SHA-2 hash functions. These are needed as new attacks on current algorithms are constantly found [79]. Brute-force attacks themselves become more feasible with computational power becoming cheaper and more abundant.

In this work we will mostly ignore specifics of cryptographic algorithms and treat encryption as a black-box such that the only way to decrypt a message is to know the correct key.

2.2 Cryptographic Protocols

Despite advances in cryptography, encryption does not provide confidentiality. It merely turns a mathematical decryption problem into a key-sharing problem. Most of the time, keys are not shared by principals ahead of time so this must be accomplished while an attacker has access to the network. As an example let us examine the basic asymmetric cryptography scenario given in Figure 2.2. If Alice wants to allow Bob to securely communicate with her she may provide her public key over the network. Bob would then encrypt his message and transmit it back. However there are two attacks that can be launched here. Firstly an attacker can easily intercept Bob's message and provide their own instead. As Alice's key was sent in plaintext they can also encrypt their message with the public key and claim to be Bob. A slightly more subtle attack is for the attacker to intercept Alice's public key and replace it with their own. When Bob encrypts his message with this key only the attacker will be able to read it. To combat such attacks we have cryptographic protocols to share keys securely as well as accomplishing other objectives such as proving authentication of one or more parties.

A cryptographic protocol is, in this context, a series of communications between two or more principals. A principal will usually be an independent communicating entity, linked to other principals through some network. These communications take place to establish one or more security related goals, or security properties. Such properties may include:

- AUTHENTICATION Some protocols aim to establish that the principals are indeed who they claim to be. A typical example would be for logging in to an

online service such as e-mail or Internet banking.

- CONFIDENTIALITY Protecting the contents of a message, such that they can only be accessed by people with the requisite access rights.

- INTEGRITY Allowing a user to validate that the contents of a message have not been accidentally or deliberately modified.

These days there are increased demands on security with the Internet becoming increasingly invaluable for significant tasks such as shopping, banking and communication. With new applications such as electronic voting there is continued development needed for new protocols.

There is a lot of interest in these protocols because, despite clever planning, flaws can still be found in them, even though it may take years for this flaw to become known. A flaw in a protocol means that an attacker can either glean secure information or mislead a principal in some way that was not intended. Indeed, Ross Anderson states in [3] that "If security engineering has a unifying theme, it is the study of security protocols".

2.2.1 Protocol Narration

There is a widely-used method of succinctly describing protocols, called protocol narrations. A typical line of one of these narrations could be:

$$A \rightarrow B : \quad A, P$$

This line simply says, principal A sends to principal B a message containing two parts, A's name and a password P. This could be part of a protocol to allow A to get access to some resource from B. If part of the message is to be encrypted, this can be represented as follows:

$$B \rightarrow A : \quad B, \{B, Msg\}_{KAB}$$

This line states that B sends a message to A consisting of its own name in plaintext and then an encrypted block which can be decrypted using the key KAB. When decrypted the message consists of two parts, B's own name again and some other message. An example of a full protocol narration is given below for the Wide Mouthed Frog protocol.

$$A \rightarrow S: \quad A, \{B, KAB\}_{KAS}$$
$$S \rightarrow B: \quad \{A, KAB\}_{KSB}$$
$$A \rightarrow B: \quad \{mess\}_{KAB}$$

Protocol narrations of this nature have a serious limitation. This method of describing a communication protocol only describes what messages are sent by each principal, omitting instructions to the receiver on what action to take upon receiving each message. Joseph E. Stoy states in [75] that "A notation is important for what it leaves out". Thus we may think that these typical protocol narrations must be extremely important as they leave out most of the inner workings. Often important complexities are only uncovered when a developer attempts to implement the protocol. Depending on the level of their expertise they may take an easy option at this point, or worse, fail to spot the problem.

A more accurate description of a protocol would include inputs, checks and how encrypted message parts are decrypted. These extra assumptions need to be represented more formally for static analysis of the protocol to be possible. To get an idea of the extra information needed to model a protocol, below is a representation of the Wide Mouthed Frog protocol taken from [12].

1.	A	$\rightarrow$		:	$A,S,A,\{B,KAB\}_{KAS}$	[assuming KAB is a new key]
1'.		$\rightarrow$	S	:	x_A,x_S,x'_A,x	[check $x_S = S$; $x_A = x'_A$]
1".			S	:	decrypt x as $\{x_B,x_{KAB}\}_{KAS}$	
2.	S	$\rightarrow$		:	$S,x_B,\{x_A,x_K\}_{KSB}$	
2'.		$\rightarrow$	B	:	y_S,y_B,y	[check $y_B = B$]
2".			B	:	decrypt y as $\{y_A,y_K\}_{KSB}$	
3.	A	$\rightarrow$		:	$A,B,\{m1,...,mk\}_{KAB}$	
3'.		$\rightarrow$	B	:	z_A,z_B,z	[check $z_B = B$; $z_A = y_A$]
3".			B	:	decrypt z as $\{z1,...,zk\}_{y_K}$	[$_{y_K}$ is bound to KAB]

This additional notation now makes clear the additional work done by the receiver of the message. The first two components of the message are the sender and then the receiver. This convention will also be followed by messages exchanged in LySa protocols later in this thesis. The protocol above still represents three messages but the processing of each message is broken down into three steps. The first represents the sender's actions; the final two steps show the receiver's actions. The second step, represented by a prime after the message number, details the first step of work the receiver performs upon receiving a message, namely checking certain parts of the message match expected values as well as binding message parts to variables. The third step

represented by a double prime details how the receiver decrypts message parts.

2.3 Attacking a Protocol

While we shall not describe every way in which a cryptographic protocol can be attacked, it is worth introducing some basic attacks in order to give a sense of the difficulty in creating a secure protocol.

2.3.1 Man in the Middle Attack

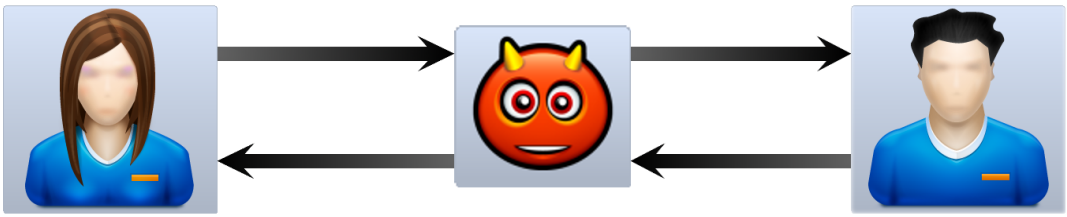


Figure 2.3: Man in the Middle Attack

The man in the middle attack is the name given to the first of the attacks described in Section 2.2. The intention of the attacker is to mislead to principals into believing they are directly talking to each other when in reality they are communicating via the attacker. In basic scenarios under tight conditions, man in the middle attacks can be foiled by using timestamps to ensure that the reply is received in good time.

2.3.2 Replay Attacks

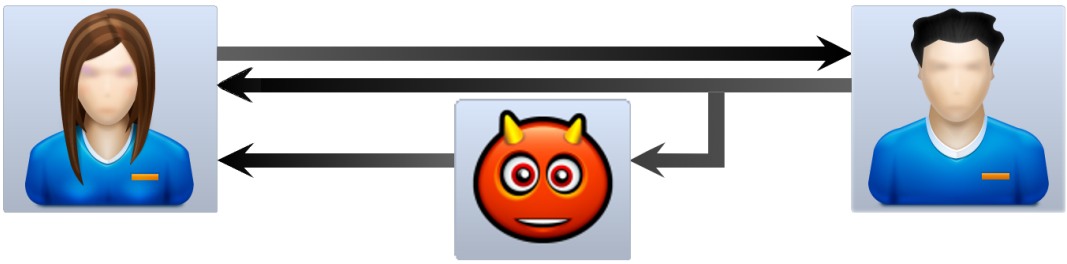


Figure 2.4: Replay Attack

A replay attack is where an attacker replays or delays a message from an earlier run of a protocol between two legitimate principals. One example may be when Alice asks Bob to provide her with his password which he dutifully does. Later the attacker masquerades as Bob by repeating Bob's password back to Alice. These attacks can be foiled by using *nonces*, a protocol primitive of a random number, or to give it the full name, a number used only once.

A more subtle use of a replay attack is where an attacker may use part of a previous protocol in a different part of a new protocol. This is known as a type-flaw attack. These rely on encrypted parts of the protocol having the same structure such that two cryptographic primitives of different types may get mistaken for each other.

A similar attack is a reflection attack. Here when a valid principal attempts to challenge an intruder to prove their identity, the attacker starts another communication with the principal in this instance issuing the same challenge to the principal and using their response to authenticate themselves. In essence, they trick the principal into authenticating themselves.

2.3.3 The Dolev-Yao Attacker

The security of communication protocols is often analysed with regard to a Dolev- Yao attacker [30]. A Dolev-Yao attacker has several abilities:

- Intercept any message sent over the network
- Decrypt any encrypted portions for which they know the correct key
- Create new messages
- Generate encrypted messages provided they have a valid key
- Send messages across the network while spoofing the sender field.

The goal of the attacker is to breach the confidentiality or integrity of the message or foil authentication mechanisms. As well as injecting messages which an attacker originates, it is also possible for an attacker to replay messages that it intercepts between legitimate principals and use any information passed in earlier communication to decrypt or alter later messages.

2.4 History of Cryptographic Protocol Analysis

Analysis of cryptographic protocols has a long history going back 20 years to BAN Logic [22]. Cryptographic protocols continue to be a widely studied field in computer security due to the ever increasing demand for secure communication and reliable authentication. Before becoming an accepted standard, protocol models are analysed, debated and re-proposed by security experts the world over. Even after this process and years of use, it is not unheard of for a new flaw to be found in the protocol which undermines its functionality. The trouble is that while the protocols themselves are usually concise, needing only a few messages to terminate, the flaws can be very subtle. The most famous example of such a flaw would undoubtedly be in the Needham-Schroeder Public Key protocol [53]. A simple attack, and fix, was presented by Gavin Lowe 17 years later [45]. Automated analysis is the most efficient method for analysing cryptographic protocol models. There are numerous techniques and tools for achieving this purpose with more devised every year [5, 37].

From the late 1970s work on modelling concurrent systems by Milner and Hoare led to the parallel development of two process calculi, the Calculus of Communicating Systems(CCS) and Communicating Sequential Processes(CSP). A key feature of both of these that would lead to descendants finding use in modelling of cryptographic protocols is that interactions between parallel processes would occur via messing passing rather than through the use of a shared memory space.

Development of these languages would lead to a variety of uses. Milner's later work on the π-calculus would be extended by Abadi and Gordon and lead to the Spi calculus[1] for reasoning about cryptographic protocols. Other uses for process calculus have been in the field of performance modelling with work such as PEPA [39]. Process calculi have more recently been used outside of computer science to model biological systems with key work on Stochastic π calculus [68] spawning much research activity.

Process calculi offer several benefits for our use. As they represent the actions of principals they are directly relatable to implementations while in other techniques such as modal logic used by BAN Logic this relationship is less clear. This makes them intuitively easier to understand to computer scientists who are already familiar with programming languages. The original advantage of being able to model a concurrent system in a concise manner by focusing only on the core actions still holds.

2.5 LySa Process Calculus

The process calculus we shall be using to model cryptographic protocols in this thesis is LySa [11, 12, 21]. This choice was made as LySa represents the actions of principals in much the same way that a computer program would implement them. Additionally the well developed, fast, efficient security analyser called the LySatool is ideally suited to providing quick feedback to developers. As LySa was expressly designed to model the actions of participants engaged in security-aware communication it does so succinctly with little extraneous information. It is similar to the π-calculus [50] with some ideas taken from the cryptographic protocol modelling Spi-calculus [1] although with two key differences. The first crucial difference is that LySa does not use the concept of dedicated channels to send messages. Instead LySa assumes there is a global medium through which all principals communicate. The developers of LySa took this approach as they believed that private channels provided a layer of privacy not matched in a real world scenario where attackers can eavesdrop or add in messages of their own.

2.5.1 Syntax

$E ::=$	*terms*
n	name($n \in \mathcal{N}$)
x	variable($x \in \mathcal{X}$)
m^+	public key
m^-	private key
$\{E_1,\ldots,E_k\}_{E_0}$	symmetric encryption under key E_0
$\{\|E_1,\ldots,E_k\|\}_{E_0}$	asymmetric encryption under key E_0

Figure 2.5: Syntax of LySa terms

The syntax of LySa terms, E, can be found in Figure 2.5. The most basic terms are values which are used to represent principal names, keys, nonces and encrypted terms. Syntactically speaking these values can be broken into three subsets: names, variables and encryption expressions. The set $\mathcal{N}$ has subsets for ordinary names such as principal names, symmetric keys and nonces and a separate set for public and private keys used to represent key pairs in asymmetric encryption. Encrypted messages are tuples of terms encrypted under a key, E_0. Both symmetric and asymmetric encryption

can be modelled and differentiated in LySa and while syntactically the key can be any expression, care must be taken, particularly with asymmetric encryption, to choose a key which allows decryption to take place.

$P ::=$	*processes*
$\langle E_1, \ldots, E_k \rangle . P$	output
$(E_1 , \ldots, E_j ; x_{j+1} , \ldots, x_k) . P$	input (with matching)
decrypt E as $\{ E_1, \ldots, E_j ; x_{j+1}, \ldots, x_k \}_{E_0}$ in P	symmetric decryption (with matching)
decrypt E as $\{\| E_1, \ldots, E_j ; x_{j+1}, \ldots, x_k \|\}_{E_0}$ in P	asymmetric decryption (with matching)
$P_1 \mid P_2$	parallel composition
$(\nu\, n) P$	name creation
$(\nu \pm m) P$	key pair creation
$! P$	replication
0	termination process

Figure 2.6: Syntax of LySa processes

The syntax of processes, P, can be found in Figure 2.6. The first process models the send portion of a synchronous communication. $\langle E_1, \ldots, E_k \rangle . P$ sends a message of k-tuples on the global communication medium and then proceeds with process P. Throughout this thesis we follow the convention of using the first two tuples in a message to be the sender and receiver respectively.

$(E_1 , \ldots, E_j ; x_{j+1} , \ldots, x_k) . P$ represents receiving a k-tuple message on the global medium. The second crucial difference between LySa and Spi is that this receive process also handles pattern-matching on the incoming message and the receipt is successful if and only if the pattern matching succeeds. The first j parts have to match with constants and if this is successful the remaining $k - j$ parts are bound to variables within the scope of the next process P.

There are two processes for decryption, one for symmetric decryption and one for asymmetric decryption. For symmetric decryption, the key E_0 must be the same as that used to encrypt. For asymmetric decryption, it must be the accompanying key in a key-pair. Clearly the same type of encryption must be used for both encrypting and decrypting the message. There is no restriction placed on which key in a key pair is used for encryption and decryption allowing LySa to be used to model both public key

encryption and private key signatures. The pattern-matching for the receive process is also used here.

$P_1 \mid P_2$ denotes two processes operating in parallel. These processes may synchronise via communication or operate completely independently. $(\nu\ n)\ P$ creates a fresh name which is restricted to the process P. This name could be used as a message, a nonce or a symmetric key. $(\nu \pm m)\ P$ generates two new names m^+ and m^- which are used for the public and private keys of a key pair. Both names' scope is restricted to P. The process $!P$ denotes an arbitrary number of copies of process P running in parallel. 0 is the nil process which does nothing and is used as a terminator.

2.5.2 Message Part Order

In order to use pattern matching to analyse a protocol, it may be necessary to reorder the contents of a message so that the parts that the receiver checks are all grouped together at the start of the message. For example in the second protocol given in [53], the third message reads

$$\mathrm{A} \rightarrow \mathrm{B}: \quad \{\mathrm{I}_A, \mathrm{A}\}_{PKB+}$$

However, in the LySa formal model, the recipient of the message (the principal designated as B) first checks that the message is sent by A and stores the incoming nonce I_A in a local variable. However, to do this the message needs to be reordered so that the nonce is the second part of the message. The LySa processes for both sending and receiving the message are as follows.

Sending	Receiving
$\langle \mathrm{A}, \mathrm{B}, \{\lvert \mathrm{A}, \mathrm{I}_A \rvert\}_{PKB+} \rangle$.	(A, B; y). decrypt y as $\{\lvert \mathrm{A}; \text{nonce} \rvert\}_{PKB-}$ in ...

2.5.3 Operational Semantics

The semantic reduction relation rules are given in Figure 2.7. The COM rule makes sure that communication only succeeds when the first j values of both the input and the output are the same. When this pattern matching succeeds the remaining k-j variables are substituted for the remaining values from the input. The three rules for decryption: SDEC, ADEC and ASIG, all follow the same pattern matching as COM but also ensure that the expression being decrypted is a valid encrypted expression using the correct key. Renaming of variables in these rules and the COM rule is captured by α-conversion which submits that the name of variables is unimportant and that terms that

can only be distinguished by differing variable names are considered equivalent. In this instance it states that a variable V_z is rewritten as x_z in the receive or decrypted process. Two rules NEW and ANEW allow processing of a process inside a restriction with the provision that the restriction operator still remains. The PAR rule states that one process may proceed without modifying the other. The final rule, CONGR formalises that the reduction rules may be applied to any process that is structurally congruent to the process found in the preceding rules. Full details of structural congruence can be found in [21].

COM

$$< V_1,...,V_k > .P_1 | (V_1,...,V_j;x_{j+1},...,x_k).P_2 \ \rightarrow \ P_1 | P_2[x_{j+1} \overset{\alpha}{\mapsto} V_{j+1},...,x_k \overset{\alpha}{\mapsto} V_k]$$

SDEC

$$decrypt\ \{V_1,...,V_K\}_{V_0}\ as\ \{V_1,...,V_j;x_{j+1},...,x_k\}_{V_0}\ in\ P \ \rightarrow P[x_{j+1} \overset{\alpha}{\mapsto} V_{j+1},...,x_k \overset{\alpha}{\mapsto} V_k]$$

ADEC

$$decrypt\ \{|V_1,...,V_K|\}_{m^+}\ as\ \{|V_1,...,V_j;x_{j+1},...,x_k|\}_{m^-}\ in\ P \ \rightarrow P[x_{j+1} \overset{\alpha}{\mapsto} V_{j+1},...,x_k \overset{\alpha}{\mapsto} V_k]$$

ASIG

$$decrypt\ \{|V_1,...,V_K|\}_{m^-}\ as\ \{|V_1,...,V_j;x_{j+1},...,x_k|\}_{m^+}\ in\ P \ \rightarrow P[x_{j+1} \overset{\alpha}{\mapsto} V_{j+1},...,x_k \overset{\alpha}{\mapsto} V_k]$$

$$\text{NEW}\ \frac{P \rightarrow P'}{(\nu\ n)P \rightarrow (\nu\ n)P'} \qquad \text{ANEW}\ \frac{P \rightarrow P'}{(\nu \pm\ m)P \rightarrow (\nu \pm\ m)P'}$$

$$\text{PAR}\ \frac{P_1 \rightarrow P_1'}{P_1|P_2 \rightarrow P_1'|P_2} \qquad \text{CONGR}\ \frac{P \equiv P''\ \ P'' \rightarrow P'''\ \ P''' \equiv P'}{P \rightarrow P'}$$

Figure 2.7: The reduction relation $P \rightarrow P'$.

2.5.4 Meta-Level LySa

Development of LySa continues[13, 35] with extensions allowing analysis of more complex scenario and attacks. The most well developed of these is a meta-level extension and the LySatool supports the analysis of it. This meta-level is used to describe the scenario in which many instances of a principal are running. Many protocols, while secure in some scenarios, are demonstrably less secure when more than one copy of a principal is running concurrently. Attackers now have the option of replaying messages

from earlier or parallel protocol sessions. In order to model this, LySa is extended with indices to names and variables. Examples of Meta-LySa can be seen in Section 5.7.

2.5.5 Authentication

Crypto-points are a vital part of LySa's ability to be used to analyse protocols. Without them the LySatool would not be able to report any violations of the protocol's authentication mechanisms. A crypto-point represents a site in the protocol where either an encryption or decryption takes place. Coupled with a crypto-point is an assertion about the origin or destination of the encrypted message. When part of a message is encrypted or decrypted, the developer can choose to specify the current location as a crypto-point. Additionally, for an encryption, the developer can then choose to specify one or more crypto-points where decryption should take place during a valid run of the protocol. This is done like this:

$$[\text{at } a \text{ dest } \{b\}]$$

The crypto-point for the corresponding decryption specifies where the message should have been encrypted and is as follows:

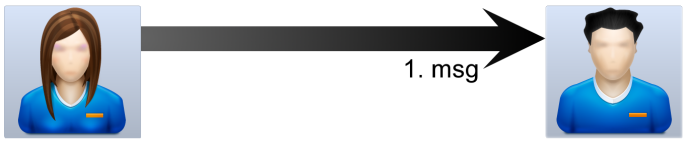
$$[\text{at } b \text{ orig } \{a\}]$$

The analysis performed by the LySatool is concerned with establishing the validity of these assertions.

2.5.6 Example LySa Models

2.5.6.1 Communication

1. A→ B: msg



This is the most basic communication protocol which is worth modelling in LySa; Alice sends a message to Bob. In the LySa model we see that there is a bit more going on than the protocol narration describes. Firstly Alice has to generate the message that she is going to send to Bob. In LySa, (ν msg) P restricts the scope of *msg* to the process P. In this case, to Alice only. After generating a fresh message, Alice transmits the message ⟨A, B, msg⟩ consisting of the name of the sender A, the intended recipient B and the message itself.

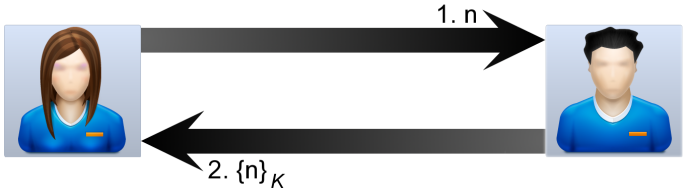
Bob, whose actions are modelled on the last line of the protocol, is waiting to receive a triple. Using pattern matching, he ensures that the first two values are *A* and *B*. If this is successful, the variable *x* is then bound to the value *msg*.

Both processes are then terminated by the nil process 0.

$$(\nu\ msg)\ \langle A, B, msg\rangle.0$$
$$|$$
$$(A, B; x).0$$

2.5.6.2 Nonce Handshake

1. A→ B: n
2. B→ A: $\{n\}_K$



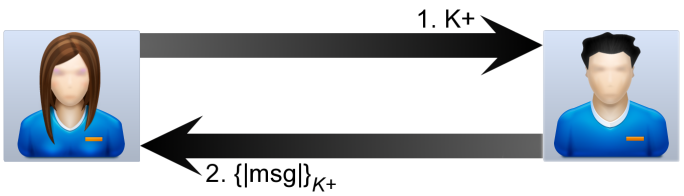
This protocol sees messages being exchanged between two principals. This protocol is used to achieve authentication of principal *B* to principal *A*. The LySa model first establishes that both Alice and Bob share knowledge of a symmetrical key *K*. In this instance the scope of *K* is over both principals. On the other hand, the nonce *n* is restricted to Alice only. Alice then sends this to Bob who then encrypts this nonce with the shared key and sends this encrypted message back to Alice. Alice then applies pattern matching on the decrypted message and providing the encrypted message is *n* the protocol is successfully terminated.

This protocol also demonstrates basic use of crypto-points for authentication purposes. There are two labels defined, one where Bob encrypts the nonce and one where Alice decrypts it. When Bob encrypts the message he states that it should only be possible to decrypt the message by Alice, and Alice states that the only person who should be able to encrypt the message she receives is Bob.

$$(\nu\ K)((\nu\ n)\langle A,B,n\rangle.(B,A;y).\ \text{decrypt y as } \{n;\}_K\ [\text{at a orig } \{b\}]\ \text{in } 0$$
$$|$$
$$(A,B;x).\langle B,A,\{x\}_K\ [\text{at b dest } \{a\}]\rangle.0)$$

2.5.6.3 Public Key Communication

1. A→ B: K+
2. B→ A: $\{|\text{msg}|\}_{K+}$



We return to the protocol first seen in Figure 2.2. This protocol has similarities to the previous although we introduce asymmetric encryption. There is no shared symmetric key so in an attempt to provide confidentiality Alice creates an asymmetric key pair and sends the public key to Bob. The private key $K-$ is kept private to Alice by the scoping rules of ν. Bob then creates a new message and sends it to Alice encrypted with the public key. Alice then decodes the message using her private key. On successful decryption Alice has the message stored in the variable y.

(ν +- K) ⟨A,B,K+⟩. (B,A;x). decrypt x as $\{|;y|\}_{K-}$ [at a orig{b}] in 0
|
(A,B;k).(ν msg)⟨B,A,$\{|\text{msg}|\}_k$ [at b dest{a}]⟩ .0

2.6 Control Flow Analysis

In analysing process calculus models an extremely effective method is to use static analysis. Dynamic testing is infeasible as it would be impractical to design test cases to model every possible execution. Static analysis means that no execution is required to determine properties of the code. As such analysis could potentially be undecidable, for reasons of efficiency the analysis is typically performed on an approximation of the program. As pictured in Figure 2.8 this approximation can either be an under or an over-approximation. Under-approximation focuses on program behaviour that must occur while over-approximation focuses on behaviour that may occur. This means that under-approximation may not report all flaws while over-approximation may occasionally give false positives, namely reporting a problem that does not exist. As we wish to make guarantees as to the absence of any attacks it is better to be over-cautious. Thus over approximation is used because it has the property that if no flaws are found then none are present. The method of static analysis used by the LySatool is also guaranteed to terminate with a low polynomial running time which makes it useful as a quick compile-time check as we desire.

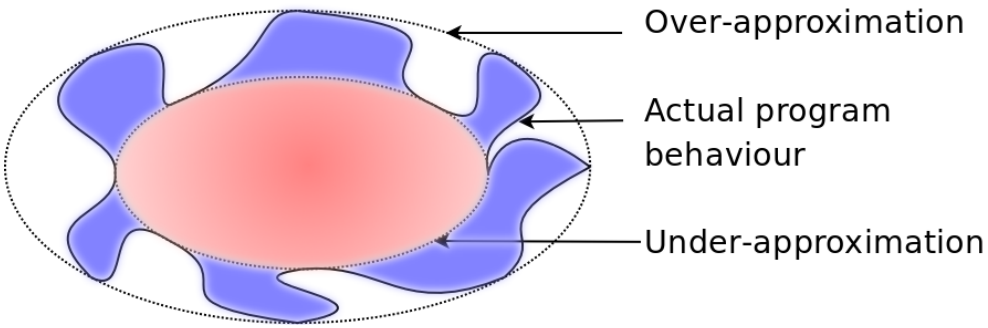


Figure 2.8: Approximation in Program Analaysis

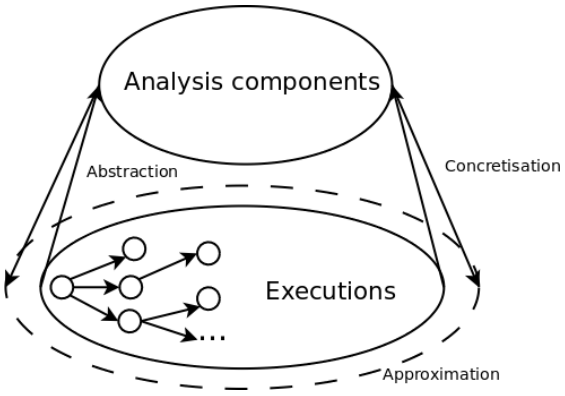


Figure 2.9: Abstraction and concretisation between semantics and analysis

The particular type of static analysis used here is control flow analysis. This is formalised in Flow Logic [56]. Flow Logic was introduced in the late 1990's and has been used for analysis of a variety of languages such as π-calculus, Spi-calculus and λ-calculus. Control flow analysis works by collecting information about a process' behaviour. This information is known as the analysis components. In an analysis of a LySa model we seek to describe the communication that a protocol represents. Thus what we need for our analysis components are two things; firstly the tuples that may have been communicated and also the values that any variables may become bound to.

A control flow analysis does not operate on an exact set of executions but a generalisation thereof. This process is known as *abstraction*. Mapping the results of this analysis back to the semantics is known as *concretisation* and as can be seen from Figure 2.9 does not match perfectly with the original process. This is due to the approximation introduced earlier. Control flow analysis has been used to discover flaws in protocols such as the Beller-Chang-Yacobi MSR protocols [6] as presented in [14].

2.7 LySatool

The LySatool[19] is an automatic tool for checking the security properties of protocols. The tool accepts as inputs protocols modelled in the LySa process calculus. It then provides feedback regarding which message parts can be decrypted as well as whether an attacker can falsely achieve authentication by inserting messages at any point. As the analysis only represents certain aspects of a process' behaviour this concretisation leads to imprecision with the originally analysed process. This over approximation can guarantee confidentiality with the downside that the tool can potentially report faults due to attacks that are impossible to reproduce in a real world scenario. Such faults have not yet surfaced in our practical examples, and to the best of our knowledge have not been reported by other users of LySa.

When reporting instances where cryptography secrecy breaks down, the LySatool uses the term 'CPDY' to signify a Crypto-Point in a Dolev-Yao attacker [30]. A Dolev-Yao attacker has several capabilities namely the ability to intercept and decrypt messages for which he knows the decryption key; as well as the ability to encrypt a message if he knows the encryption key, and finally to send messages claiming to be from a legitimate principal. To represent that an attacker can decrypt an encrypted message, the LySatool will list under the violation of authentication properties:

(a, CPDY)

Here 'a' is a valid crypto-point, specified by the developer of the model. Decryption of an encrypted message is a violation of secrecy between the legitimate principals. Additionally an attacker may be able to encrypt some message which will then be decrypted by a principal as part of a protocol. If a legitimate principal decrypts a message encrypted by an attacker it constitutes a violation of authentication as the legitimate principal will believe that only another legitimate principal can encrypt this message. This will show up as a violation represented by:

(CPDY, b)

The LySatool takes a LySa process and generates a formula in alternation-free least fixpoint logic in clausal form(ALFP) [54]. ALFP is an extension of Horn clauses such that it additionally allows existential and universal quantification in pre-conditions; negated queries; disjunctions of preconditions and conjunctions of conclusions. Details of the conversion from LySa to ALFP can be found in [20]. This formula is then

Values that may not be confidential n, $\{l\bullet, Lmess\}_{LK}$, n•, m•+, m•-, B, A, $\{l\bullet, l\bullet\}_{l}\bullet$ [at CPDY]
Violation of authentication properties (Ψ) *No violations possible*

Figure 2.10: Example LySatool Results

solved by the Succinct Solver [55]. No counter-examples or attack narrations are provided if a violation is found, as may be expected with model-checking for example, although the faster execution speed provided with the LySatool makes up for this. The results that the LySatool does report are provided in an HTML file presented similar to the example in Figure 2.10.

2.8 Protocol Security and Program Security

Until recently protocol verification was completely disconnected with protocol implementations. Verification of protocol models allows a developer to be certain that a specification of a protocol is secure, however to be of any use a developer must then implement the model. Additionally they would have to learn the language and tools which would allow them to analyse a formal model of their protocol. Even when working with an already validated specification it is not unheard of for errors to creep in at the implementation stage.

Historically few applications required secure communication capability so the development of this could be overseen by experienced experts. These days however even most console games boast online connectivity. We cannot assume that the development of this area is overseen by people with the necessary knowledge to avoid security pitfalls. The risk is increased as developers are often trying to create implementations from incomplete specifications. These often take the form of protocol narration perhaps with supporting information although implementation details are often left out. Such gaps are often left to developers to fill and this can lead to interoperability problems or more serious failures. Developers without experience are more likely to make bad decisions at such points.

Additionally these days it is much easier to develop applications. The average

developer has access to integrated development environments, managed code and automatic memory management. Added to this is access to the Internet, a huge, if not always accurate, repository of code and help from thousand of unverified sources. In fact, many IDEs will help write programs for you with auto-completion of keywords and method identifiers. All of this makes it easier than ever to start writing applications. There is also more demand for applications with mobile devices from several manufacturers having their own market-place where individual developers can have their applications downloaded by thousands of people. While this explosion of developers has many upsides, an unfortunate consequence has to be that with more developers come more mistakes. Additionally inexperienced developers are likely to repeat mistakes of the past or perhaps copy code from other sources propagating any errors in this code as in [57]. Developers face numerous pressures such as the requirement to deliver on time and to make sure that the delivered product performs as expected. Security is often seen as an optional extra compared to these other pressures. Rarely will a developer unfamiliar with security take the time to properly familiarise themselves with the nuances of their task or take the time to learn a formal method that may help them implement a secure deliverable. For these reasons, we believe that analysis of implementations is just as important as analysis of specifications.

Chapter 3

Increasing Accessibility to Process Calculi

In recent years, programming languages have benefited from becoming more user-friendly. Integrated development environments (IDEs) give developers access to auto-complete functionality, syntax checking and debugging information in an easy to understand way. In contrast, process calculi often seem to revel behind their veil of secrecy. IDEs do have some disadvantages; they can encourage bad programming practices and sloppy, even lazy coding due to developers being used to their IDE fixing their mistakes. Despite this there are many advantages that users of process calculi such as LySa could benefit from. The notion of cryptographic protocol development assisted by an IDE's auto-complete suggestions is worrying because inexperienced developers may choose to follow these suggestions without understanding the protocol they are constructing. Despite this, allowing a developer to instantly check their syntax is correct is a powerful tool and much more useful than receiving possibly confusing syntax errors at the time of analysis. More than that, by helping inexperienced developers use such languages we encourage more people to utilise these powerful but intimidating languages.

The LySa Toolkit in Eclipse (LyTE) was designed to help developers work with the LySa code generated by Elyjah. It aims to make LySa slightly less intimidating to any developers who were introduced to the process calculus through Elyjah. Like the other tools presented in this thesis it is implemented as an Eclipse plugin so much of the operation will be familiar to developers who should not have to learn how to use a new tool in order to learn a new language. A screenshot of LyTE running in Eclipse is presented in Figure 3.1. In this Eclipse configuration, LySa code is written and edited

in the LySa Editor in the top left portion of the window, features such as the syntax highlighting visible in the screenshot are described in Section 3.2. In the top right, is the animation produced by the VisuaLySa as described in Section 3.4. The bottom half of the screenshot shows the AnaLySa console and features output from both the AnaLySa and the LySatool. More information on the AnaLySa is in Section 3.3.

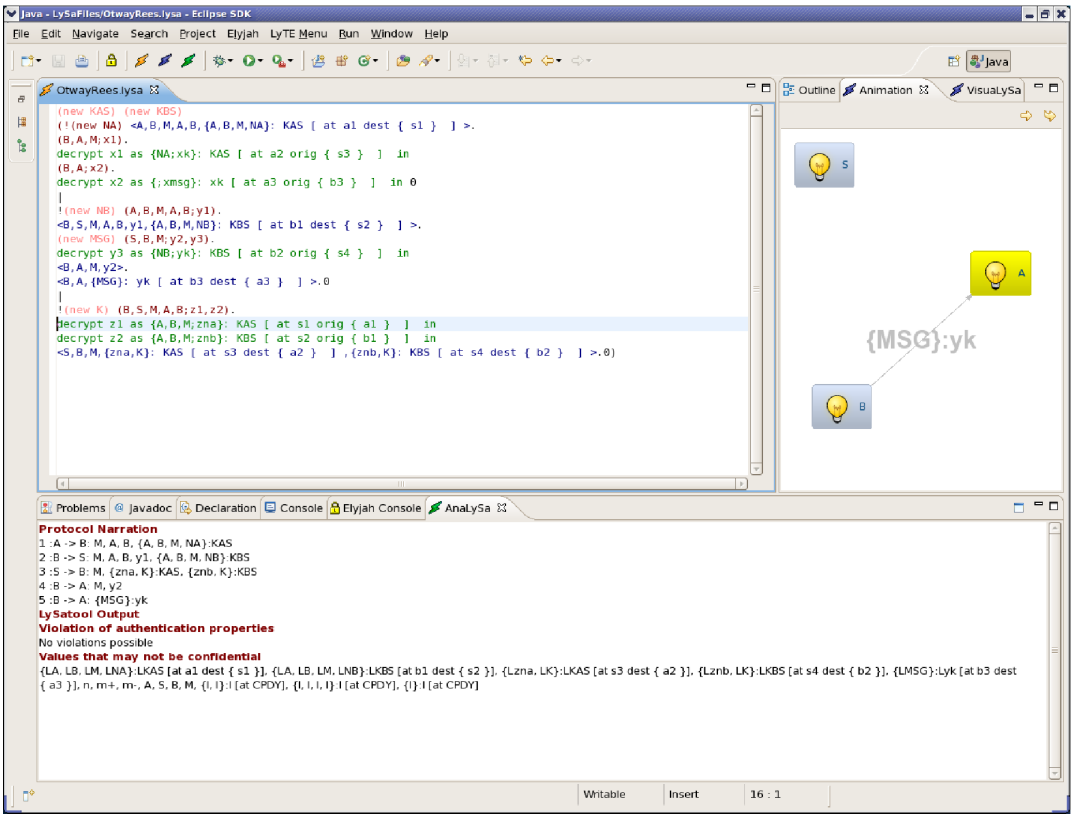


Figure 3.1: The LySa Toolkit in Eclipse

3.1 LySa Parser

The following sections as well as the Hajyle tool in Chapter 6 required a custom LySa parser. Throughout this work we used the JavaCC parser generator. However in this tool a parser is not sufficient, a means of navigating an abstract syntax tree is also required. JavaCC by itself will only return whether an input file is successfully parsed. There is an add-on for JavaCC known as JJTree which allows the generated parsers to produce syntax trees. In order to generate a parser to do this there are several stages of computation. First a JJTree file is converted to a JavaCC file which is then converted

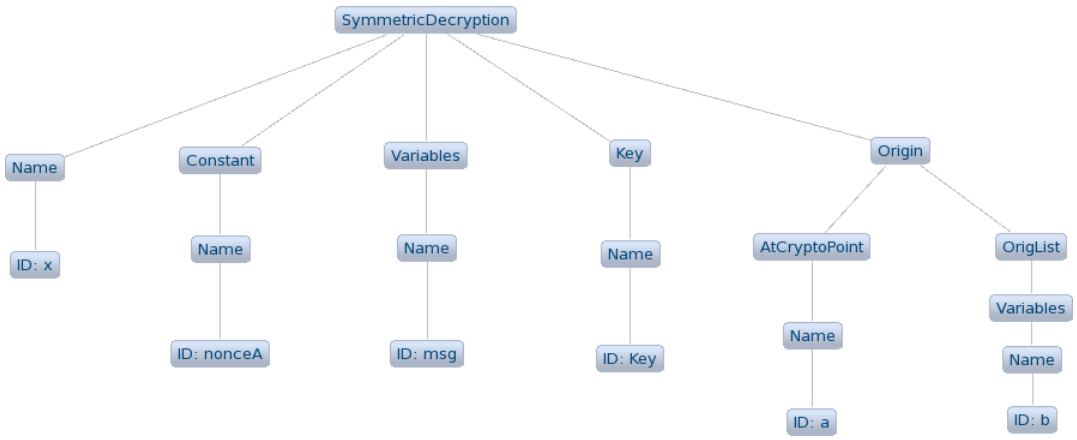
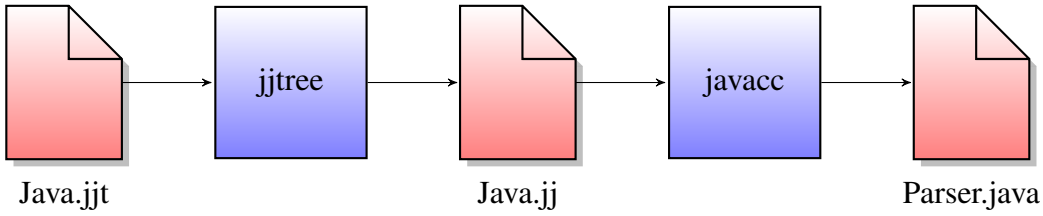


Figure 3.2: Syntax Tree of decrypt x as {nonceA; msg}:Key [at a orig {b}]

to a Java source file before finally being converted to a class file by the Java compiler.



JJTree defines a Java interface, `Node`, which all parse tree nodes must implement. This interface provides methods for navigating through a tree of nodes using methods to traverse to the parent as well as through the children of a node. This interface is implemented by a `SimpleNode` class which is created automatically by JJTree and can then be extended or modified as needed. Here the class was modified to increase the error handling capabilities and change the return values of the tree traversal methods from `Node` to `SimpleNode` to eliminate the need for casting. Using JJTree, after a source file is parsed the root of the abstract syntax tree is returned as a `SimpleNode`. An example of a LySa process represented as an abstract syntax tree is given in Figure 3.2.

We frequently have to search the source code for certain keywords or method invocations. This is performed by conducting a depth-first search on a block.

```
public static void methodName(SimpleNode node){
    if (node.toString().equals(nodeName)) {
        // Process Node
    } else {
        for(int i = 0; i < node.jjtGetNumChildren(); ++i) {
```

```
                    SimpleNode n = node.jjtGetChild(i);
                    if (n != null) {
                        methodName(n);
                    }
                }
            }
}
```

Many of the methods that require searching an abstract syntax tree will take the form of the above template. This method is called on the root of some block of code, such as a single line, a method, a switch block or a whole class. The current node is then checked to see if it contains the required string, `nodeName` in the above example. If so then some additional processing will be performed, otherwise this method is called on the children of the node. The values generated by methods of this sort are usually stored in static `ArrayList` or `HashMap` objects allowing the original method to create `Iterator` objects to iterate through these values.

3.2 LySa Editor

The LySa Editor is designed to help developers write and edit LySa models. It features dynamic parse checking to identify errors quickly and provide the developers with a useful error message and suggested fixes. Fitting in to the Eclipse platform, the error is underlined, the line marked, and an error message added to the problem console. This checking can identify syntax flaws such as using the wrong punctuation, misspelt key words or missing pattern matching on receive and decrypt processes. Below is an example of such checking; the keyword “decrypt” is misspelt and the error is underlined and an error icon signifies that a problem is on this line.

```
(!(new NA) <A,B,M,A,B,{A,B,M,NA}: KAS [ at a1 dest { s1 }  ] >.(B,A,M;x1).
decript x1 as {NA;xk}: KAS [ at a2 orig { s3 }  ]  in  (B,A;x2).
decrypt x2 as {;xmsg}: xk [ at a3 orig { b3 }  ]  in 0
```

The editor also provides syntax highlighting, which helps separate send, receive and decrypt processes by highlighting them in different colours. There is additional static analysis which checks that for each send process there is a corresponding possible receive process with the same source and destination as well as the right number of tuples. This is summarised below.

```
findSendReceiveCryptoPoints(SimpleNode node) {
  if (node == SendProcess)
```

```
    key  = source + "," + dest + "," + # of msgParts
    value = LySa representation of SendProcess
    send.put(key,value)
  if (node == ReceiveProcess)
    key  = source + "," + dest + "," + # of msgParts
    value = LySa representation of ReceiveProcess
    send.put(key,value)
  if (node == EncryptTerm)
    if (node.atCryptoPoint != null)
      encryptAt.add(node.atCryptoPoint);
    if (node.destCryptoPoint != null)
      if (node.hasMultiDestCryptPoint()){
        for node.destCryptoPointChild : destCryptoPart
          destCrypto.add(destCryptoPart)
      else
        destCrypto.add(node.destCryptoPoint)
  if (node == Decryption)
    if (node.atCryptoPoint != null)
      decryptAt.add(node.atCryptoPoint)
    if (node.origCryptoPoint != null)
      if (node.hasMultiOrigCryptoPoint)
        for node.origCryptoPointChild : origCryptoPart
          origCrypto.add(origCryptoPart)
      else
        origCrypto.add(node.origCryptoPart)

  for node.child : n
    if (n != null)
      findSendReceiveCryptoPoints(n);
}
```

Firstly the abstract syntax tree for the LySa file is searched for all send and receive processes. When either of these are found, an entry is added to a HashMap which has a key comprised of the source, destination and number of remaining tuples in the message and a value of the reconstructed LySa process. Additionally when there are encrypt terms or decryption processes various Sets are appended with the labels given for the current location cryptopoint, with separate sets for encryption and decryption,

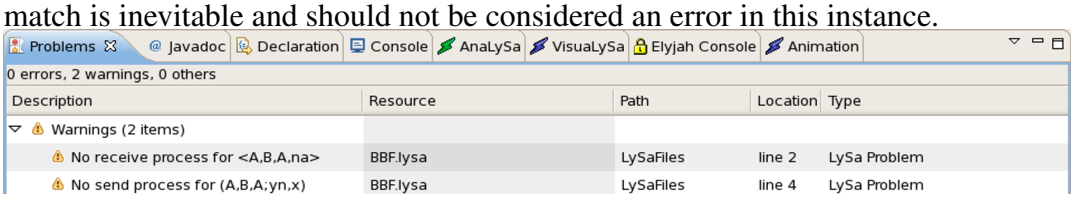
and the destination or origin cryptopoint.

```
findErrors(SimpleNode root){
  findSendReceiveCryptoPoints(root)

  decryptAt2 = decryptAt.copy()
  encryptAt2 = encryptAt.copy()

  encryptAt.removeAll(origCrypto)
  origCrypto.removeAll(encryptAt2)

  decryptAt.removeAll(destCrypto)
  destCrypto.removeAll(decryptAt2)

  for send.keys : sendKey
    if (!receive.contains(sendKey))
      reportError (send.get(sendKey))

  for receive.keys :receiveKey
    if (!send.contains(receiveKey))
      reportError (receive.get(receiveKey))

  for encryptAt : encAt
    reportError (encAt)

  for decryptAt : decAt
    reportError (decAt)

  for origCrypt : orig
    reportError (orig)

  for destCrypto :dest
    reportError (dest)
}
```

Once the file has been parsed, errors are generated under certain conditions. The key set for the send HashMap is iterated over and a check is made to make sure that there is a corresponding key in the receive HashMap. If this is not true, an error is generated

for the send process indicating that there is no corresponding receive process. The same process is applied to the receive key set and any errors indicate that the receive process has no corresponding send process.

This provides a weak form of liveness checking that warns users of possible areas of deadlock in their models. Where this is not the case, a warning is flagged on the send and/or receive process in question and added to the list of warnings in the Eclipse Problems console. Below we see that processing a LySa model has identified two such warnings. In this instance they are caused by a mismatch between a send and receive process with an extra variable mistakenly added to the receive process in the second warning. The reason such differences are flagged as warnings and not errors is that there exists situations where message source or destinations use variables so a mismatch is inevitable and should not be considered an error in this instance.



Additionally matching crypto-point labels from encryptAt and origCrypto sets and decryptAt and destCrypto sets are removed. Any labels that remain in these sets indicates a failure in the cryptopoints, with errors generated for any of the following reasons:

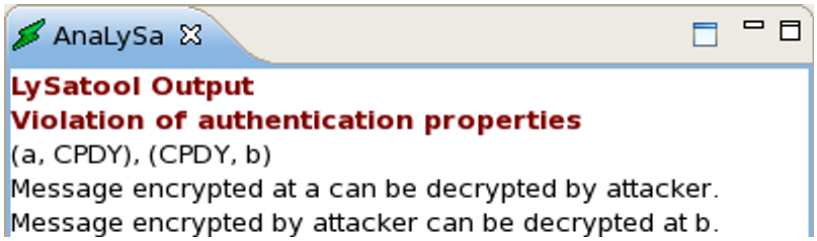
- No matching decrypt crypto-point (Decrypt process lists different origin)
- No matching encrypt crypto-point (Encrypt process lists different destination)
- Specified destination crypto-point not found
- Specified origin crypto-point not found

The user's LySa model can then be analysed with the LySatool, the results of which are displayed in an Eclipse console. In order to best integrate the results into Eclipse, a custom version of the LySatool has been developed. This version replaces the standard output functionality of the LySatool which generates three HTML files. Instead, the analysis results are output in a series of structured text files which can then be read and parsed into Eclipse. A user can then choose which sections of the analysis they want displayed. The three sections are:

- Violations of crypto-point assertions

- Message parts which may not be confidential
- Tree Grammar

All three of these analysis parts reveal potential different protocol flaws. Violations of crypto-points reveal errors where a Dolev-Yao attacker can either decrypt a message from a principal or masquerade as a legitimate principal and construct an encrypted message which a legitimate principal decrypts, believing it to come from another legitimate principal. They also reveal potential errors where two encrypted messages have the same message format and can be swapped. When analysing Meta-LySa models, it also reveals any assertions that do not hold within a single session revealing possible replay attacks. Rather than just the crypto-point pairs presented in Section 2.7 our Eclipse plugin additionally provides a textual representation such as:



Unlike syntax errors or liveness checking these violations are reported in a console used purely for LyTE purposes.

Message parts which may not be confidential reveal which message parts can be read by a Dolev-Yao attacker. The list of these message parts are printed in the LyTE console. A developer has the option of additionally specifying key message parts and receiving an additional alert if they are reported as not being confidential. These keywords are to be included in a LySa model in a comment as a comma separated list following the phrase “keywords:”. For example if the LySa model included the line:

/* KEYWORDS:message,secret, msg */

Then if any of the message parts were reported as not being confidential, these would be flagged with an additional error warning.

The Tree Grammar results show the mapping between variables and the values that they become bound to during a process execution. This allows us to see when variables are mapped to incorrect values. This analysis result is perhaps more advanced than reporting which message parts are not confidential and the violations of crypto-point assertions. Accordingly the protocol errors which can be identified by these results are more subtle than those reported by the other analysis results.

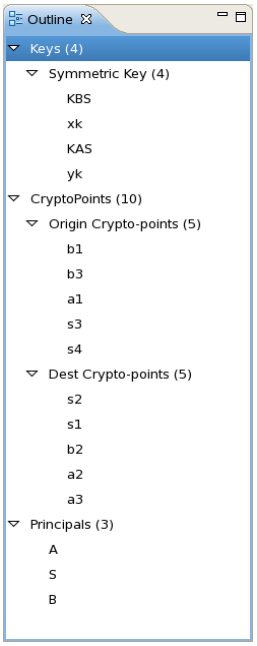


Figure 3.3: The LySa Editor Outline View in Eclipse

3.2.1 Editor Outline

Eclipse provides the opportunity to display relevant information about a file in what is called an Outline View. This is used to provide a structured outline of the file. The information provided is different for each editor. For example, with the standard Java editor, the Outline View shows a hierarchy of the Java classes, methods and fields. As LySa does not have much of a hierarchical structure beyond the separation into principals, we decided to use the Outline View for the LySa Editor to provide pertinent information about the protocol. The information provided is:

- Names of principals
- Names and types of keys
- Names and types of crypto-points

As can be seen in Figure 3.3, this information is presented in a structured, hierarchical system which divides keys into symmetrical and asymmetrical keys, asymmetric keys are further divided into public and private keys. Principal names are not further sorted although crypto-points are also divided into origin and destination sub groups. Keys are identified only by the name of the variable used in the encryption or decryption process. We do not additionally show any aliases that may be accumulated during

messages sent or created during ν processes. This is a conscious decision which makes it possible to identify exactly what variable names are used for encryption and decryption. This allows a developer to make a quick check as to which variable names are used.

3.3 AnaLySa

The AnaLySa tool translates a LySa model into a standard protocol narration. Standard protocol narrations are often used to describe protocols, although the trouble is that they only contain half of the protocol. Namely they do not say what the recipient does upon receiving a message. However, they are used because they succinctly describe the messages exchanged. Figure 3.4 shows a typical protocol narration, namely the Otway-Rees protocol. This provides a succinct overview of the protocol and provides a developer with quick 'sanity check' of a LySa model.

1. $A \rightarrow B : M, A, B, \{A, B, M, N_A\}_{K_{AS}}$
2. $B \rightarrow S : M, A, B, \{A, B, M, N_A\}_{K_{AS}}, \{A, B, M, N_B\}_{K_{BS}}$
3. $S \rightarrow B : M, \{N_A, K\}_{K_{AS}}, \{N_B, K\}_{K_{BS}}$
4. $B \rightarrow A : M, \{N_A, K\}_{K_{AS}}$
5. $B \rightarrow A : \{SECRET\}_K$

Figure 3.4: Otway-Rees protocol narration

In order to translate from a LySa model to a protocol narration such as the Otway-Rees protocol in Table 3.1 we parse each principal in turn and pick out both the send and receive processes. Each of these is stored in an individual queue data structure for each principal as dramatically depicted in Figure 3.5. In this diagram we use the syntax "→B [M, A, B, $\{A, B, M, NA\}_{KAS}$]" to denote a message sent to principal *B* which has as its contents the message parts enclosed in square brackets. The inverse "A→ [M, A, B, y1]"syntax represents a receive process. In Figure 3.5 the processes are colour-coded to represent the matching send and receive processes for each message.

```
createQueue(Principal p, Node node){
  if (node == SendProcess)
    queues.get(p).addSend(node)
```

(ν KAS) (ν KBS)
(!(ν NA) ⟨A,B,M,A,B,$\{$A,B,M,NA$\}_{KAS}$ [at a1 dest { s1 }] ⟩
.(B,A,M;x1). decrypt x1 as $\{$NA;xk$\}_{KAS}$ [at a2 orig { s3 }] in
(B,A;x2). decrypt x2 as $\{$;xmsg$\}_{xk}$ [at a3 orig { b3 }] in 0
|
!(ν NB) (A,B,M,A,B;y1).
⟨B,S,M,A,B,y1,$\{$A,B,M,NB$\}_{KBS}$ [at b1 dest { s2 }] ⟩.
(ν SECRET) (S,B,M;y2,y3).
decrypt y3 as $\{$NB;yk$\}_{KBS}$ [at b2 orig { s4 }] in
⟨B,A,M,y2⟩.
⟨B,A,$\{$SECRET$\}_{yk}$ [at b3 dest { a3 }] ⟩.0
|
!(ν K) (B,S,M,A,B;z1,z2).
decrypt z1 as $\{$A,B,M;zna$\}_{KAS}$ [at s1 orig { a1 }] in
decrypt z2 as $\{$A,B,M;znb$\}_{KBS}$ [at s2 orig { b1 }] in
⟨S,B,M,$\{$zna,K$\}_{KAS}$ [at s3 dest { a2 }] ,$\{$znb,K$\}_{KBS}$ [at s4 dest { b2 }] ⟩.0)

Table 3.1: LySa model of Otway-Rees Encryption

```
      analyseSend(node, name)
    if (node == ReceiveProcess)
      queues.get(p).addReceive(node)
    for node.child : n
      if (n != null)
      createQueue(p, n);
}
```

While creating the queues for each principal, the first process is examined to find a send process. This will be used to construct the first message in the protocol. In our example this is "1: A→B [M, A, B, $\{$A, B, M, NA$\}_{KAS}$]". The next task is to identify the corresponding receive process, in this case the first process in principal B's queue. In other work we will then use this information to create a mapping between the names used to refer to message parts by different principals. For the AnaLySa tool however we are merely identifying the message order so this step is not needed. While we know that the first message is from *A* to *B*, we need to check whether *A* then immediately sends another message or themselves wait for a message from another

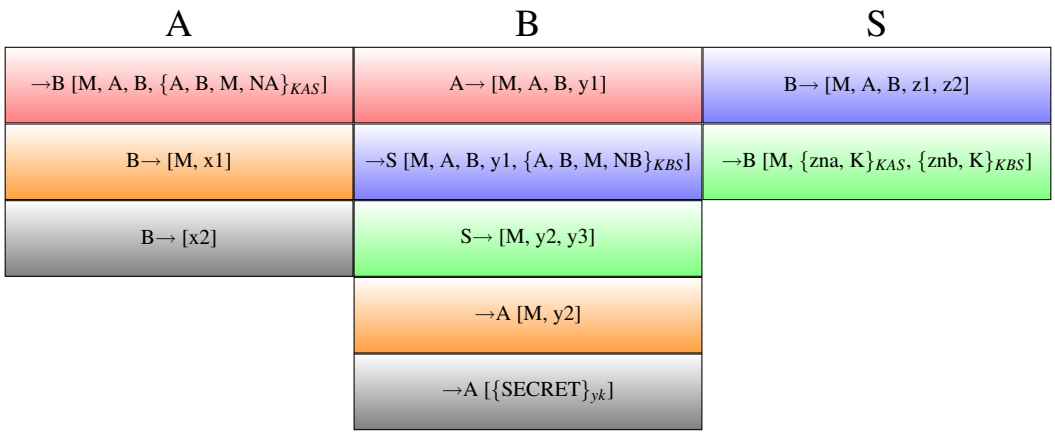


Figure 3.5: Expanded Protocol

principal. To determine this we examine the second process in *A*'s queue. If this is another send process then this becomes our next message. We would then repeat this, looking at subsequent processes if they continue to be send processes. Otherwise, as in this example, we switch to looking at the queue for the matching receive process. Popping this message off the queue, we can then examine the next process. If there are no other messages already sent and the protocol is not yet over then, as in this case, the next process will be a send process. In this instance, this is a message sent to principal *S*. We continue in this manner to determine the order of all messages including the two messages that principal *B* sends at the conclusion of the protocol.

```
for Principal : p {
  Queue q = neq Queue()
  queues.add(p, q)
  createQueue(p, p.rootNode)
  if q.first = next
        first = p
}

if (first != null){
  source = first
  dest = first

  while (true) {
    source = dest
```

```
    if (queues.get(source).isEmpty())
      break

    qp = queues.get(source).getNext()
    dest = qp.getDestination()
    if (queues.get(dest).isEmpty())
      break

    qp2 = queues.get(dest).getNext()
    processMessageContents(qp, qp2)

    if (queues.get(source).hasNext()) {
      while (queues.get(source).getNext().isSend()) {
        qp = queues.get(source).getNext()
        dest = qp.getDestination()
        qp2 = queues.get(dest).getNext()

        processMessageContents(qp, qp2)

        if (queues.get(source).isEmpty())
          break
      }
  }
}

processMessageContents(QueuePart qp1, QueuePart qp2){
  received_To_Send.putAll(qp1.getMsgParts, qp2.getMsgParts)
  Message m = new Message(counter++, qp1.principal,
        qp2.principal, qp1.getMsgParts);
  messages.add(m)
}
```

Further LySatool integration within LyTE is accomplished by highlighting message parts which appear in the list of the LySatool's "Message Parts that may not be Confidential". We ignore from the analysis results any encrypted sections so if there are any nested encryptions this won't reveal whether an attacker will be able to read the encrypted inner section but whether they will be able to read the base message

Figure 3.6: AnaLySa in Action

parts. A screenshot of the AnaLySa running on the Otway-Rees protocol is shown in Figure 3.6.

3.3.1 Elyjah/Hajyle Translation Validation

The AnaLySa can be used to act as a simple form of Translation Validation [65] for the Elyjah tool presented in Chapter 5 and the Hajyle tool in Chapter 6. The output from the AnaLySa can be compared with the execution trace generated by running the JaLAPI implementation which is detailed in Section 5.3.6. Both versions of the protocol narration should have the same structure. This acts as a case-by-case check of the tool's translation. It is said that there are two ways to prove the correctness of tools like Elyjah and Hajyle, either proving the tool or proving the output. Using the AnaLySa allows the developers to perform a simple proof of the correctness of the LySa model in relation to the Java implementation. Figure 3.7 shows how using two representations of a protocol we can reveal different properties that are satisfied by the protocol. By using the LySatool on a LySa representation of the protocol we can reveal the security properties of the protocol. Possessing a Java implementation of a protocol we can also test for properties such as liveness, that the intended messages were indeed shared and that parties are authenticated as intended. In Chapter 4 we continue this by examining the performance properties of a protocol. In this figure, we see how reducing both Java and LySa representations to a common alternative depiction allows us to reason about the equivalency of these differing representations.

Here we present the execution trace from the Java implementation that generated the LySa model of the Otway-Rees protocol used in Section 3.3.

```
18-Jul-2008 11:12:09 Network send
```

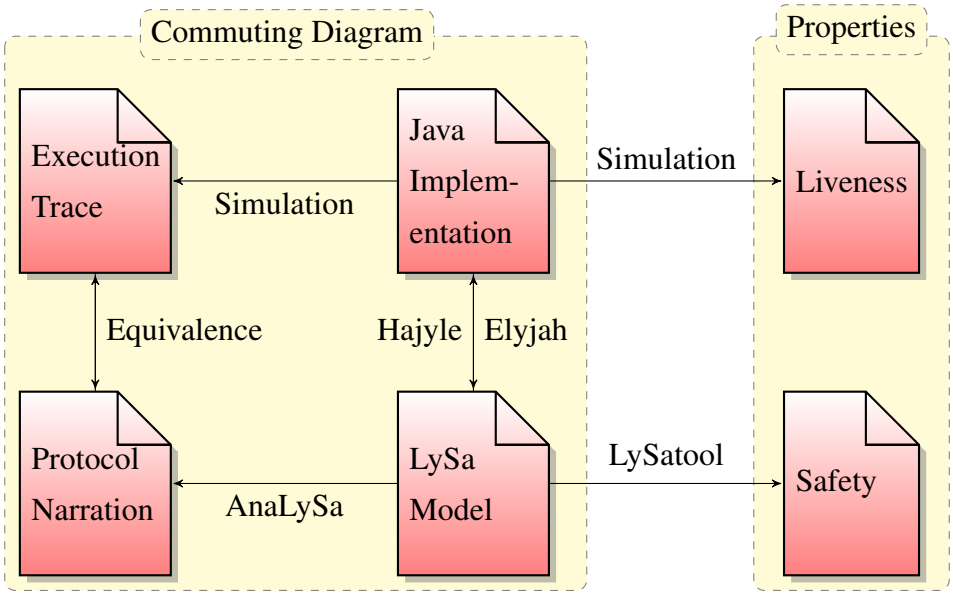


Figure 3.7: Using LyTe to validate Elyjah and Hajyle translation

```
INFO: 1 :A -> B: 1, A, B, {M0, M1, M2, M3}:KAS
18-Jul-2008 11:12:09 Network send
INFO: 2 :B -> S: 1, A, B, 124AFog5PqCr/HCsdZ/5Jg==,
        {M0, M1, M2, M3}:KBS
18-Jul-2008 11:12:09 Network send
INFO: 3 :S -> B: 1, {M0, M1}:KAS, {M0, M1}:KBS
18-Jul-2008 11:12:09 Network send
INFO: 4 :B -> A: 1, qirLTAOPhn5FmkkY70GBtN8rIUSLgJCf
18-Jul-2008 11:12:09 Network send
INFO: 5 :B -> A: {M0}:yk
```

Comparing the two different protocol narrations reveals the same structure thus providing evidence of correct translation for this example. However, there are a few key differences. Firstly the serial number, *M* in the AnaLySa result, is replaced with the actual number in the execution trace. Use of encryption results in two further differences, firstly the execution trace only contains the structure of the encrypted section and secondly the fourth tuple in the second message and the second tuple in the penultimate message are the encrypted string rather than the variable names used to identify them. This system has a couple of limitations. Firstly as we are comparing standard protocol narrations we are only checking that the messages that are sent are the same. There

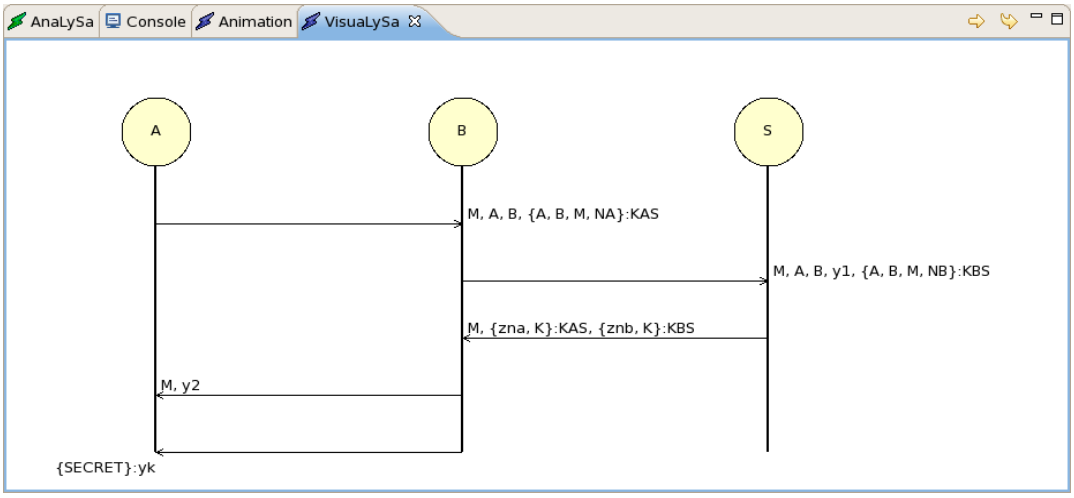


Figure 3.8: VisuaLySa Sequence Diagram

are no guarantees that the message is processed the same way in the LySa version as in the original implementation. Further work on this tool could correct this shortcoming. At the moment however, we believe it provides a useful reassurance to developers that Elyjah or Hajyle is correctly translating their protocol representation.

3.4 VisuaLySa

As an alternative to the protocol narration provided by the AnaLySa, we additionally present graphical representations of LySa models. This option is useful as a teaching tool for explaining cryptographic protocols to people unfamiliar with them, and beyond this audience to developers who are starting to use LySa and while still getting used to the syntax, want to learn through experimentation. The VisuaLySa provided by LyTE offers two forms of outputs. The first is a sequence-diagram style picture as seen in Figure 3.8. This is a simple pictorial depiction of a protocol but distinctly shows the messages sent in the correct order and we easily see the flow of information during the protocol. For teaching purposes, the diagram can be generated one message at a time to enable explanation of complicated protocols and at any point the diagram can be exported for inclusion in lecture notes or other publications.

A more interactive and intuitive view is also provided as a protocol animation. This view simulates the protocol narrations given earlier in this thesis such as in Section 2.5.6. On initiation of the VisuaLySa, the correct number of principals is generated. The user then has the option of stepping through the protocol one message at

a time or viewing the protocol in its entirety. Messages move across the screen from one principal to another and accumulate over a protocol run to give a lasting protocol narration. LySatool integration is provided by the depiction of a Dolev-Yao attacker. This provides a user with a way of seeing the attacker's knowledge accumulating over the length of a protocol. Crypto-point assertion violations are also represented as part of the animation. Messages that can be decrypted are shown as an additional message sent from the sender to the attacker. Any encrypted portions that can be forged by the attacker are depicted as a message moving from the attacker to the intended recipient. After a protocol run has been exhausted, the picture shows the messages that have been received by each principal and also displays which communication links between principals have been utilised. The user also has several options for controlling the speed of the animation. There is a user-adjustable slider to control the speed of the message movement. There is an additional option to stop the message halfway between principals.

These stages are depicted in Figure 3.10. In the first picture a message is being sent from principal B to principal A. Although the motion that is present in the animation cannot be easily depicted the link between the two principals is an arrow showing the direction of information flow. This is in fact the final message in the Otway-Rees protocol and as such the attacker has accumulated an extensive amount of knowledge, all from merely observing the information sent across the network. The second picture shows the situation after the message has arrived at A. The attacker now also has knowledge of the encrypted secret but does not have the ability to decrypt it and learn the secret. The final picture shows a succinct pictorial summary of the protocol. We can see from this that principals A and S never directly communicate.

In order to demonstrate what happens when a protocol goes wrong we introduce an error into the protocol. In the fourth message we mistakenly send yk instead of $y2$ to principal A. Figure 3.9 shows a message originating from B toward the attacker revealing that the attacker can decrypt the message. There is an additional message from the attacker towards principal A denoting that the attacker can create an encrypted message which A believes to be from B and successfully decrypts it.

3.5 LyTeX

Another feature included in the LySa Toolkit in Eclipse is called LyTeX. This tool converts a LySa model into a typeset model suitable for inclusion in a LaTeX document.

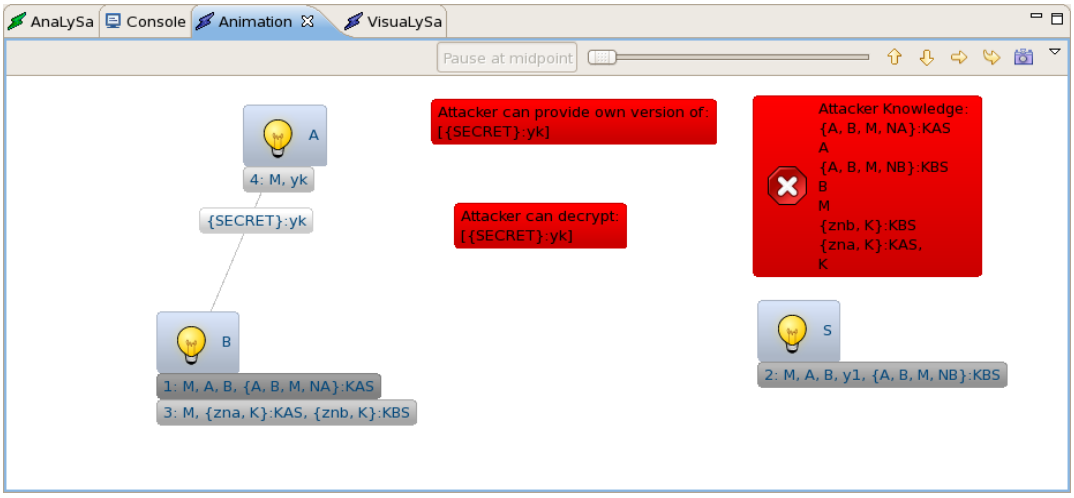


Figure 3.9: VisuaLySa Animation with attacker

All the LySa models in this thesis were generated using this tool having been written in the LySa Editor which was used to check the syntax of these models prior to their inclusion in this thesis.

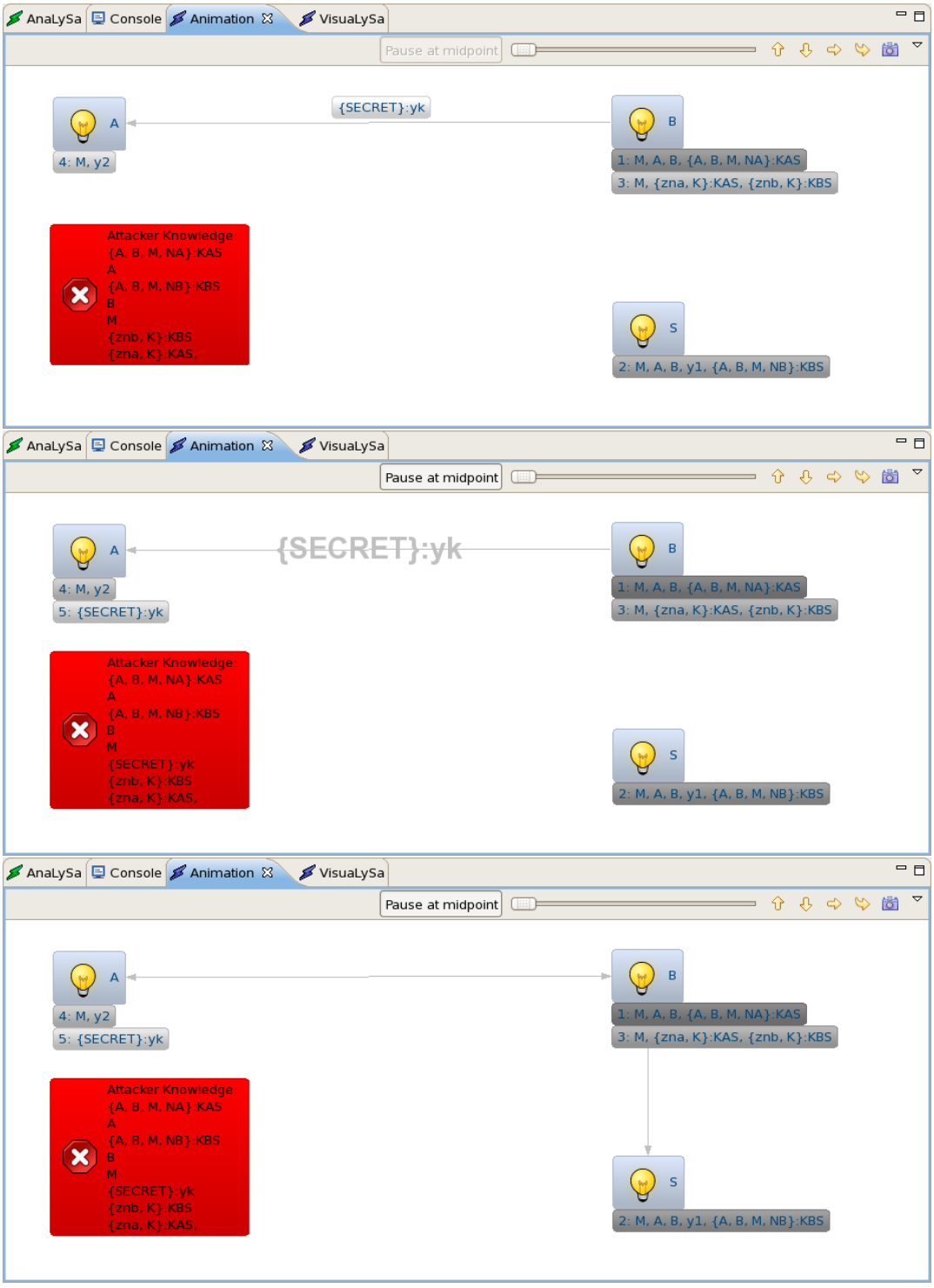


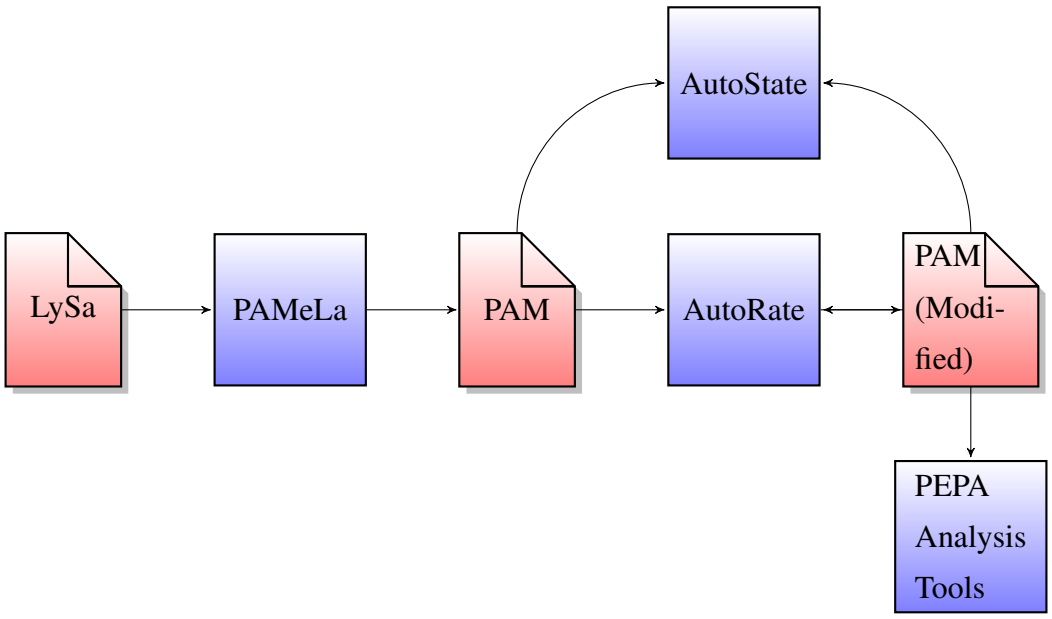
Figure 3.10: VisuaLySa Animation

Chapter 4

Concerning Performance Driven Cryptographic Protocol Development

While security and privacy are looked upon as fundamental rights, the speed at which messages are delivered must also be viewed as a critical matter. Noted security expert Bruce Schneier repeatedly states in [71] that "Security is a trade-off". While people are willing to wait or undertake extra work for security, there are limits to how long they will do so. There are multiple methods for analysing the security properties of cryptographic protocols[22, 1, 10] and equally many methods of analysing the performance of computer systems[16, 39, 2]. In this section we explore merging the two fields by developing tools to analyse the run-time performance of a cryptographic protocol. It is a worthwhile goal to determine if in individual scenarios a balance can be determined for these two competing properties of a protocol. We do not discuss using these techniques to find potential security issues such as identifying stale keys or denial of service attacks although further work could potentially expand on the tools presented here to accomplish such goals.

In this chapter we present a way to leverage existing analytic techniques developed for the PEPA calculus. In Section 4.1 we describe the new representation used to allow performance modelling and in Section 4.2 we detail the process to convert LySa models into this new format. In Section 4.4 and Section 4.5 we present two tools which allow us to visualise and improve our performance model and in Section 4.6 we show the sort of analysis that can be performed on these models. The relationship between the various tools and file formats that this section deals with is detailed in the following diagram.



4.1 PAMeLa

PAM, Process Algebra Modelling, is a framework for representing process calculi as labelled continuous-time Markov chains and analysing them using analysers developed for the PEPA calculus[39]. PAM allows stakeholders in Markovian process calculi to generate the underlying labelled transition system from their favourite Markovian process calculus and then pass this transition system to the PEPA Eclipse Plug-in for solution and visualisation of the results. The PAM language represents transition systems using XML allowing the structure of the transition system to be easily represented. This work is part of the PEPA Eclipse plug-in as featured in papers such as [78].

We present PAMeLa, short for PAM-ersatz-LySa, an Eclipse plug-in which translates LySa models into PAM files which we can then analyse. The translation process between LySa and PAM is straightforward for simple protocols but as the protocols get larger subtleties are discovered. A PAM file contains a series of states which can have several transitions. A transition is a description of the resulting state along with a label and rate for the transition from the original to the new state. For our use, a state is a composition with each process representing a principal in a protocol. As cryptographic protocols do not typically have a notion of choice the transitions in a single process are linear. There is some work to be done with automatically unrolling multiple and nested encryption. Additionally, although there is no choice in a single principal there is some option in the transitions. The excerpt below is taken from the PAM representation of the Otway-Rees protocol. In this example, either the first or second process can tran-

sition to a new state, although the third is waiting for an accompanying send process from principal B for this receive process.

```
<state>
 <composition>
  <process>(new NA)</process>
  <process>(new NB)</process>
  <process>(B,S,M,A,B;z1,z2)</process>
 </composition>
 <transitions>
  <transition>
   <composition>
    <process>(A,B,M,A,B,(A,B,M,NA))</process>
    <process>(new NB)</process>
    <process>(B,S,M,A,B;z1,z2)</process>
   </composition>
   <via name="(new NA (by A))" rate="n_A"/>
  </transition>
  <transition>
   <composition>
    <process>(new NA)</process>
    <process>(A,B,M,A,B;y1)</process>
    <process>(B,S,M,A,B;z1,z2)</process>
   </composition>
   <via name="(new NB (by B))" rate="n_B"/>
  </transition>
 </transitions>
</state>
```

The current stage of the protocol this excerpt represents is emphasised in Table 4.1.

The full state space diagram can be seen in Figure 4.1. These diagrams can be automatically generated by the PAMeLa Eclipse plug-in, although this one has been hand drawn in an attempt to provide readability. The diagram demonstrates that while each principal is a linear system, there are several areas where different principals can advance at different speeds before meeting up on shared transitions.

(ν KAS) (ν KBS)
(!(ν NA) ⟨A,B,M,A,B,{A,B,M,NA}$_{KAS}$ [at a1 dest { s1 }] ⟩.
(B,A,M;x1). decrypt x1 as {NA;xk}$_{KAS}$ [at a2 orig { s3 }] in
(B,A;x2). decrypt x2 as {;xmsg}$_{xk}$ [at a3 orig { b3 }] in 0
|
!(ν NB) (A,B,M,A,B;y1).
⟨B,S,M,A,B,y1,{A,B,M,NB}$_{KBS}$ [at b1 dest { s2 }] ⟩.
(ν SECRET) (S,B,M;y2,y3).
decrypt y3 as {NB;yk}$_{KBS}$ [at b2 orig { s4 }] in
⟨B,A,M,y2⟩.⟨B,A,{SECRET}$_{yk}$ [at b3 dest { a3 }] ⟩.0
|
!(ν K) (B,S,M,A,B;z1,z2).
decrypt z1 as {A,B,M;zna}$_{KAS}$ [at s1 orig { a1 }] in
decrypt z2 as {A,B,M;znb}$_{KBS}$ [at s2 orig { b1 }] in
⟨S,B,M,{zna,K}$_{KAS}$ [at s3 dest {a2}], {znb,K}$_{KBS}$ [at s4 dest {b2}] ⟩.0)

Table 4.1: LySa model of Otway-Rees Encryption

Different rates are generated for encryption, decryption, message generation and communication for each separate principal and link between them. Using the PEPA Eclipse Plug-in we can then determine if the protocol is suitable for the intended deployment scenario and if we were determined to improve one part of the infrastructure which part it should be. In cases such as the above Otway-Rees protocol, preconditions on the protocol such as the shared keys KAS, KBS are not included in the PAM model as we are attempting to represent a typical protocol run and such pre-conditions would only occur once.

4.2 LySa to PAM

We use the same parser we developed for the LySa Toolkit in Eclipse introduced in Section 3.1. The first task is to identify all the keys used in the protocol. This is performed by examining all encrypt and decrypt processes and storing the name of the variable used as a key in the process along with the type of encryption used, symmetrical or asymmetrical. This process is similar to the `findSendReceiveCryptoPoints` method defined in Section 3.2. We must then find all additional names that these vari-

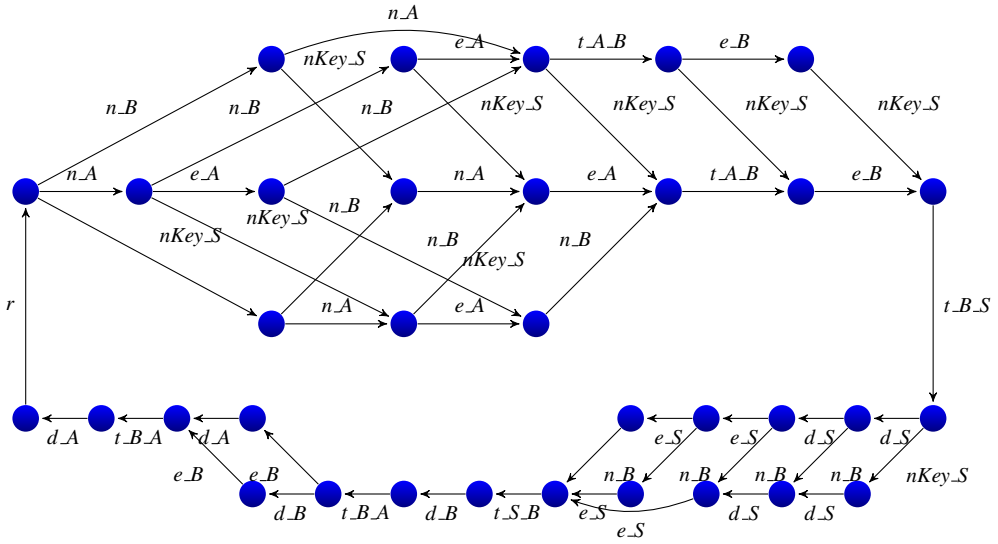


Figure 4.1: State Space Diagram of the Otway-Rees protocol

ables are known by in other principals. We do this by matching the send and receive processes of messages as in Section 3.3. We can then create a mapping between the variable names which are used in a receive statement with the names used in the corresponding send process.

```
findAllKeys(){
  findKeys(root)
  foreach keySet : k
    if (received_To_Send.contains(k))
      keySet.add(received_To_Send.get(k))
}

findKeys(SimpleNode node) {
  if (node == EncryptTerm || node == Decryption)
    keySet.add(node.key)

  foreach node.child : n
    if (n != null)
      findKeys(n);
}
```

We then analyse each principal's section of the protocol in turn. To do this we first identify the roots of the principals by searching for the nodes marked "Principal" in

LySa	PAM State	Transition Name	Rate
($\nu\pm$ K)	(new+- K)	(new+- K (by A))	nPair_A
(ν K)	(new K) (new K)	(newKey K (by A)) (new K (by A))	nKey_A n_A
$\langle$A,B,x$\rangle$|(A,B;x)	(A,B,x)|(A,B;x)	transfer (from A to B)	t_A_B
$\langle$A,B,$\{$x$\}_K$[at a1]$\rangle$	(A,B,(x))	encrypt (at a1)	e_A
$\langle$A,B,$\{$|x|$\}_K$[at a1]$\rangle$	(A,B,(|x|))	encrypt (at a1)	ae_A
decrypt x as $\{$;z$\}_K$[at b1]	decrypt x as $\{$;z$\}$:K	decrypt (at b1)	d_B
decrypt x as $\{$|;z|$\}_K$[at b1]	decrypt x as $\{$|;z|$\}$:K	decrypt (at b1)	ad_B

Table 4.2: PAM Transitions

the abstract syntax tree generated by the LySa parser. Each of the principal's process definitions are then analysed in order to generate a queue of transitions from one state to another. A summary of the transition names and rates used in a PAM representation of a LySa protocol are given in Table 4.2.

Each item in the queue has three parts, the current state, the name of the transition needed to move to the next process and the rate ascribed to the transition. The current state is typically represented by the LySa representation of the process. This rule's exception is in the case of encrypted message parts in a send process. The problem in this situation is that LySa does not have a separate process for encryption but considers it as a base term. For performance modelling terms it is imperative to construct additional processes for each encryption step. For this reason we introduce a new notation where nested parentheses represent plain-text message parts to be encrypted and the standard LySa curly braces represent encrypted segments. It is worth mentioning that to avoid issues of confusion with the XML format we replace the angled brackets denoting send with standard parentheses.

The simplest process to handle is the LySa new key-pair process. From the abstract syntax tree, we reconstruct the appropriate LySa model. This acts as our state name. The name of the transition is similar to this but appended with which principal is taking the action. The rate name takes the form nPair_ along with the principal name.

If the process is the more general creation process, then we first determine if the new item we are creating is a key or an object such as a nonce or another such message part. This is due to key generation being a more complicated task which therefore is deserving of a different rate. We have already created a list of variables used as keys

and created a mapping of variables sent between principals to determine other names associated with these keys. We check whether the variable in the LySa process is used as a key or sent to another principal and used as a key there. This gives the two options seen in Table 4.2.

Creating the PAM transitions for a LySa send process involves creating intermediate states for encryption before sending a message. The first task is to step through and analyse each message part. If the message part is plain-text then this is stored and if the message part is an encrypted block we store a version of the block in an unencrypted form and also a version with the LySa formatted encryption. Once all message parts have been processed in this manner we focus on creating the intermediate encryption steps. Each encryption is treated as an individual step including nested encryption. The name of the current state is the LySa model of the send process with the current encryption step and any as-yet unencrypted segments enclosed in parentheses. The name of the transition takes the form "encrypt (at CryptoPoint)". The rate of the transition depends on the type of encryption. If it is asymmetric decryption the rate is of the form "ae_Principal" otherwise the rate would be "e_Principal". Once the message is fully encrypted the final transition is to send the message. The current state is the fully encrypted message. The name and rate of the transition are of the same form as the name and rate for a receive process. This allows us to find appropriate pairs in the next stage. An example of a queue for the LySa excerpt ⟨A, B, {x}:K [at a dest {b}]⟩ is given below.

	State	Transition Name	Transition Rate
1.	(A, B, (x))	encrypt (at a)	e_A
2.	(A, B, {x}:k)	transfer from A to B	t_A_B

Translating a receive process into the appropriate PAM transition properties is simple. The current state property is the LySa process reconstructed. The name of the transition takes the form "transfer from Source to Destination". The rate of the transition is similar and has the form t_Source_Dest.

A decrypt process has the LySa code for the current state and the transition name is of the form "decrypt (at CryptoPoint)". The rate is either ad_Principal or d_Principal depending on whether the decryption was asymmetric or symmetric.

For each principal we now have a list of transitions from one state to the next with a name and a rate for the transition to the next state. The task is to convert these multiple queues into a single PAM state transition file. We use a custom object to

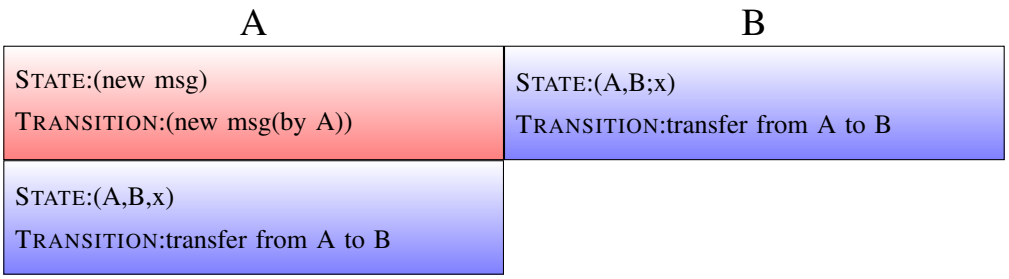


Figure 4.2: Example Transition Queues

represent a PAM state with a List to represent a composition and a Set of these for possible transitions. Our initial state is constructed by taking the first transitions from each principal's queue. To determine the possible transitions that can be made from this state we examine each queue in turn. We check to see if the transition name is a transfer. If the transition is something else, such as an encryption or message part generation we can construct a new state by keeping all other queues the same and advancing this one. If the transition is a transfer then we can only advance if there is a corresponding transfer on another principal. When we have checked each principal in turn we then process these new states.

4.2.1 Example

We can demonstrate this process on the example below. There are two principals which have already had their individual transition queues created. Together they represent the LySa process:

$$(\nu\ \text{msg})\ \langle \text{A,B,msg} \rangle.0$$
$$|$$
$$(\text{A,B;x}).0$$

Taking the transition queues in Figure 4.2 we construct the current state by taking the current state from the transition at the front of each queue. In PAM format that is represented as follows:

```
<state>
 <composition>
  <process>(new msg)</process>
  <process>(A,B;x)</process>
```

```
  </composition>
```

We now have to create the transitions for this state. To do this we look at the transition name property. Principal A has the transition name "(new msg (by A))" so does not require a match. We can then, temporarily, remove this transition from the queue and construct the new state using the transition beneath, this gives us:

```
  <transitions>
   <transition>
    <composition>
     <process>(A,B,msg)</process>
     <process>(A,B;x)</process>
    </composition>
```

We then need to append the name and rate of the transition we removed, in this case:

```
    <via name="(new msg (by A))" rate="n_A"/>
   </transition>
  </transitions>
 </state>
```

We now replace the transition we just removed and attempt to add any more transitions. In this case however, the only other principal has a receive with no matching send process so can not advance, therefore we move onto the new state just created.

```
 <state>
  <composition>
   <process>(A,B,msg)</process>
   <process>(A,B;x)</process>
  </composition>
```

At this point when trying to create a transition we see that both principals' next transition is "transfer from A to B". As these are matching send and receive statements, both principals can progress.

```
  <transitions>
   <transition>
    <composition>
     <process>0</process>
```

```
     <process>0</process>
    </composition>
    <via name="transfer (from A to B)" rate="t_A_B"/>
   </transition>
  </transitions>
 </state>
```

At this point all that is left to create another state with this process and a transfer to the original state as a reset.

4.3 Asynchronous Message Transfer

In the previous section we described a translation that features synchronous message transfer. We now present an alternative approach that models asynchronous communication. As neither of these is necessarily right or wrong we provide both approaches to the user and allow them to choose.

To model messages in transit we introduce the notion of a buffer which is modelled by a queue of transitions like the principal transition queues. In order to create the PAM representation of asymmetric communication we add an extra step in the translation. If we have a "transfer" process and do not find a matching process on any other principal's queue then we firstly deduce whether we are dealing with a send or receive process. Due to the same brackets being used for both processes in PAM this is achieved by looking for the presence of a semi-colon which signifies the end of pattern matching in a receive process.

When dealing with a send process we add the message to the buffer and advance the current protocol queue. The prefix of the transition name is changed from transfer to send and likewise the rate prefix from t to s. We need a new rate for this as the transfer is now in two parts so asymmetric communication requires a separate rate or else will be modelled as taking twice as long as it does.

If we have a receive process we compare this to the first element in the buffer for a match. If the two transitions have the same transition name we remove the transition from the buffer and advance the protocol with the transition name of the form receive (by Dest from Source). This then adds the following two possible PAM transitions to those listed in Table 4.2.

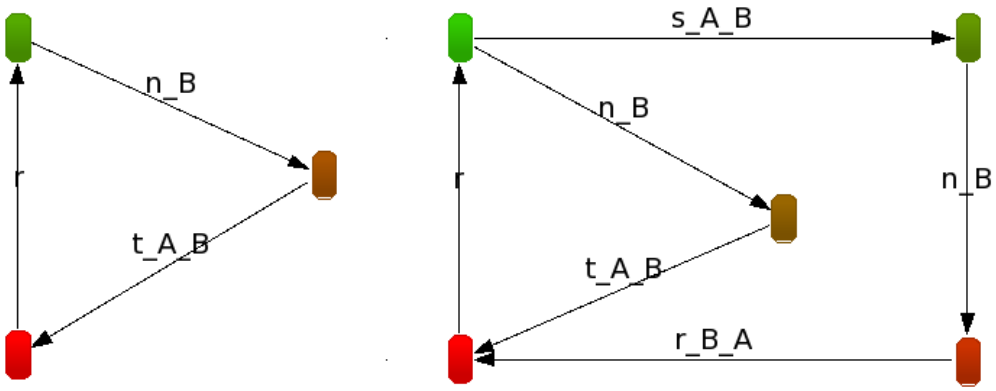


Figure 4.3: Comparison of Synchronous Only Vs Additional Asynchronous Communication

LySa	PAM State	Transition Name	Rate
⟨A,B,x⟩	(A,B,x)	send (by A to B)	s_A_B
(A,B;x)	(A,B;x)	receive (by B from A)	r_B_

4.3.1 Example

To show the difference between synchronous communication and the new asynchronous alternative presented in this section we examine an extract of a protocol.

$$\langle A,B,msg\rangle.0$$
$$|$$
$$(\nu\ Z)\ (A,B;x).0$$

Using synchronous communication there is only one possible protocol flow. Principal *B* must generate the new variable *Z* before a transfer can take place. If we assume asynchronous communication however, then principal *A* can send the message without principal *B* being ready to accept it. Principal *B* can then perform the ν process followed by the receive process. Figure 4.3 show state transition diagrams of both of these options, synchronous communication is shown on the left and asynchronous on the right. As shown in this diagram, asynchronous communication includes the synchronous option of the instantaneous send and receive transfer process.

The state space expansion viewed here is not a cause for much concern. In a more realistic example such as the Otway-Rees protocol, with synchronous message transfer,

the state space as seen in Figure 4.1 has 34 states. With asynchronous communication this expands to 46, an expansion of a little over 35%. This does not significantly affect the time taken to analyse the performance model.

4.4 AutoState

Figure 4.1 shows an example of the state transition system in pictorial form. While this form of viewing a protocol can be illuminating, the author can personally reassure the readers that constructing diagrams such as this is time consuming to do so by hand. We thus present a tool which takes a PAM file and provides a similar diagram to that in Figure 4.1.

We use the Zest package previously used in the VisuaLySa tool in Section 3.4. This visualisation package for Eclipse plugins easily allows graph nodes to be re-arranged by the end-user. With complicated protocols the graphs are often non-planar so can be confusing to read. Allowing a user to highlight and manipulate nodes and edges helps to minimise this confusion.

Another advantage that the diagrams our tool creates have over images such as Figure 4.1 is that we add colour to the nodes. The intitial state is coloured green; the final state is coloured red; intermediate states take on colours between green and red. This shading from green to red shows the progress of the protocol. There will be a single transition from a red state to a green one marking the re-initialisation of the protocol.

Figure 4.4 shows a screenshot of the AutoState-produced diagram representing the same state-transition diagram shown in Figure 4.1. As you can see although the transitions are labelled, the names of the state are not shown to save space. The state name can however be shown by moving the mouse over the node. The screenshot shows the very clear change in colour as the protocol progresses in a clockwise direction from the left-hand side.

4.5 AutoRate

Being able to express a LySa model as a state transition system, either as a PAM file or being represented in pictorial form reveals certain aspects of the protocol. For example, from Figure 4.1 we can identify where there are diverging paths and the key points where these paths join again. Despite this, the greatest amount of knowledge is gained

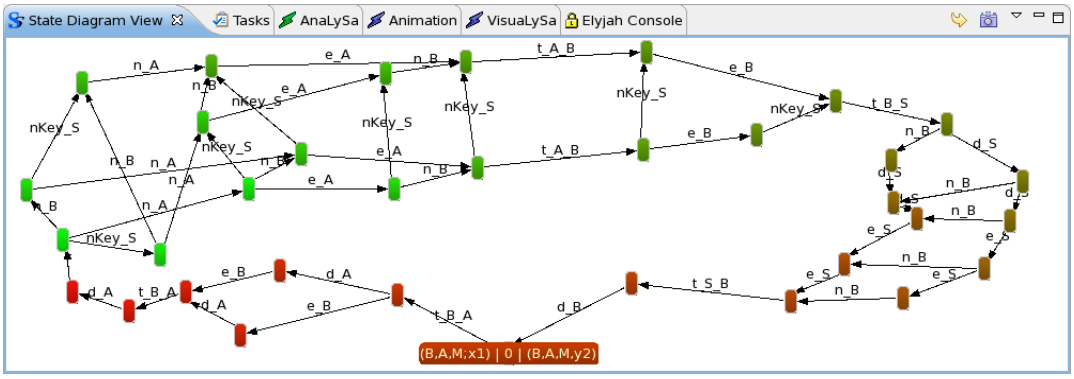


Figure 4.4: AutoState Screenshot

through performing numerical analysis on a PAM file. For this to be successful, the user needs to provide values for the rates for each transition. Providing accurate rates is seen as a key problem in performance analysis. Different functions such as encryption and decryption, not to mention variations such as symmetric or asymmetric cryptography take different times on the same device. For useful and reliable information to be gleaned from the analysis users would need to spend a lot of time determining the appropriate values. Equally for our domain, protocols may be running over different devices and different communication mediums and accurate rates would be needed for each of these variations.

We solve both of these dilemmas by providing a tool which takes a user's description of a deployment scenario and provides the rates that fit this description. This is done with a graphical interface that makes it easier for those unfamiliar with the formal model to achieve desirable results quickly. A screen-shot of this can be seen in Figure 4.5. A user can choose the type of device that the principal will run on and the network speed of the connection between communicating principals. This information will then be used to automatically calculate the rates for all the transitions.

4.6 Analysis Opportunities

Having a protocol expressed as a PAM file allows certain mathematical analysis to take place. These analysis methods were developed for the PEPA language. In this section we discuss how these techniques can be used to analyse cryptographic protocols and what they reveal about the underlying protocol.

The first tool that is ported from the PEPA Eclipse Plug-in is a single-step navigator

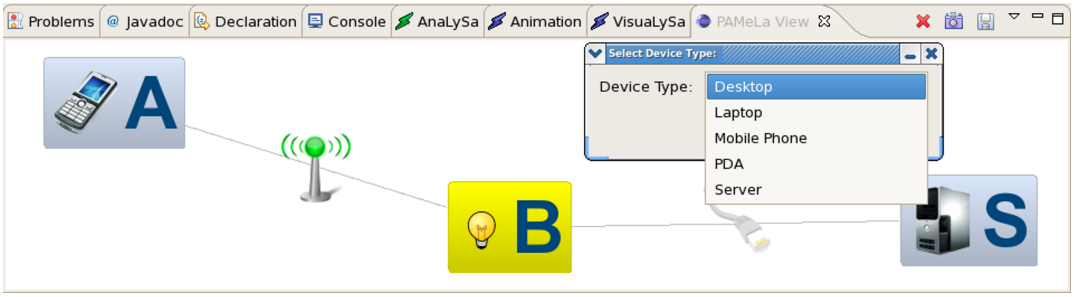


Figure 4.5: AutoRate Screenshot

for navigating through the state-space diagrams presented in Section 4.4. A screenshot of this tool in action is shown in Figure 4.6. The tool is divided into three sections. The middle section shows the current state. The top section shows all states that can move into the current state. In the first column it shows the action taken to move into the current state. The bottom section shows the possible states that can be accessed from the current state. As before, the first column shows the name of the action needed to complete the move. Highlighted in red are the changes this action makes to the current state. For example, in Figure 4.6 the first possible transition in the list is for principal A to construct the nonce NA. This changes principal A's state to the construction of a message with an encrypted segment. Other options are for principal B to create a nonce or principal S to create a key.

4.6.1 Utilisation

The first analysis option available to a user is to view the utilisation of a principal during a protocol run. This information is presented in pie-chart form for each individual principal. Examples of this sort of diagram are in Figure 4.7. The information presented here allows a user to identify potential bottlenecks in the protocol. When combined with the usage of the AutoRate tool to model deployment scenarios this form of analysis allows a developer to see which device or transmission link is the bottleneck in that particular scenario. This potentially allows the developer the chance to upgrade the equipment that is slowing the whole protocol run down.

In Figure 4.7 utilisation graphs for both synchronised and asynchronous communication types are given for the Otway-Rees protocol. It is notable that the pie-charts for principals A and S have zero or minimal difference between the different communication methodologies while principal B shows a marked difference. The pie charts

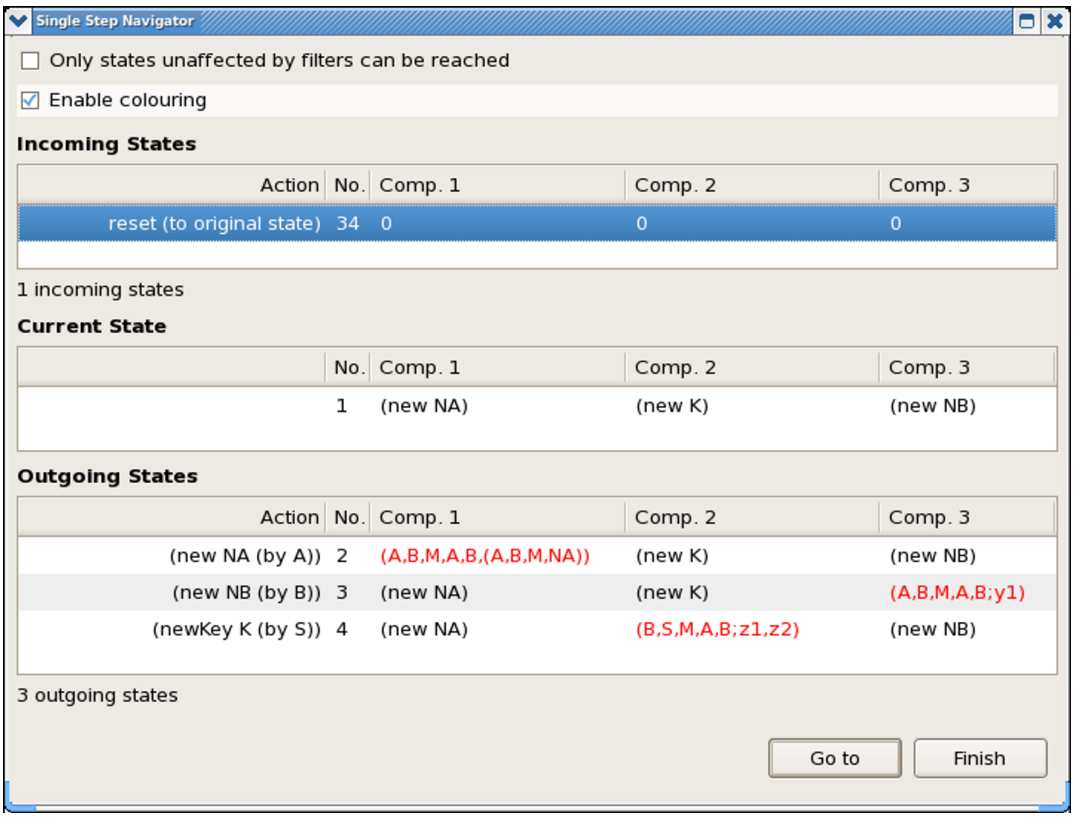


Figure 4.6: Single Step Navigator

representing principal B make it clear that changing the communication type changes which process dominates the utilisation.

In the synchronous transfer, the majority of time is spent in B's initial send process to principal S while with asynchronous message communication the dominant process is the subsequent receive process from principal S. As the rates ascribed to various transitions are not changed it seems clear that this change means that the differences in B's proportional usage is dependent on the interaction with S and that in synchronous communication principal B is waiting for S to be ready to receive the message rather than the act of communication itself taking a long time. As the only preceding process is the new key process it must be this process that is slowing principal B. This seems reasonable as the act of creating a key is sufficiently more complicated than creating a regular variable and with a sufficiently fast communication medium could still be an active process at this point in the protocol. This realisation leads us to believe that if were to look at improving the speed of the protocol we may want to examine the newKey rate for principal closely.

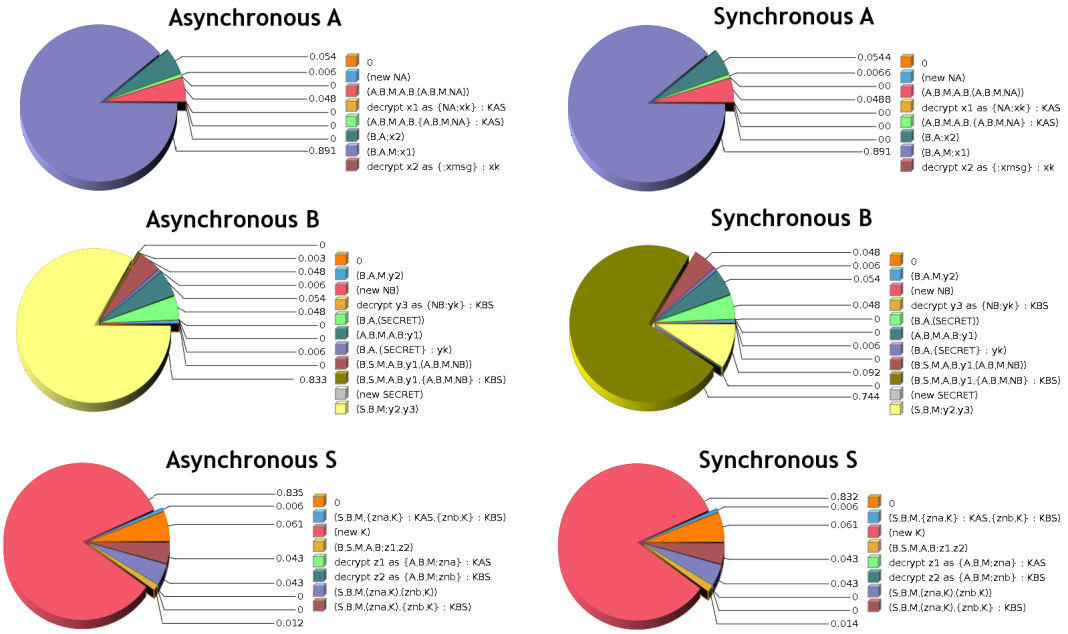


Figure 4.7: Utilisation Graphs

4.6.2 Experimentation

In the previous section we explored the possibility of using this suite of tools to determine the protocol element most in need of upgrading to improve the speed of a protocol run. In this section we use another form of analysis to determine how much resource should be dedicated to upgrading a single element. Graphs can be generated showing the throughput of a particular action versus the rate. The user can determine the values for the rate via a comma separated list or by supplying lower and upper bounds. The examples in Figure 4.8 shows typical examples of such a graph. There is a point in most graphs of this type where increasing the rate has a negligible impact on the throughput. At this point there are other bottlenecks in the system that are restricting the protocol. This allows a developer to optimise the upgrades by not spending resources where they are not getting full payback on the expenditure.

Figure 4.8 shows experimentation graphs where we compare the throughput when the t_B_S,e_S and nKey_S rate is varied in the Otway-Rees example used throughout this section. With the current rates it is clear that improving the nKey_S rate would make a much larger improvement to overall throughput than improving either the t_B_S or e_S rate would. In this scenario it may mean spending time and effort to develop a more efficient implementation of the new key method or using a different algorithm

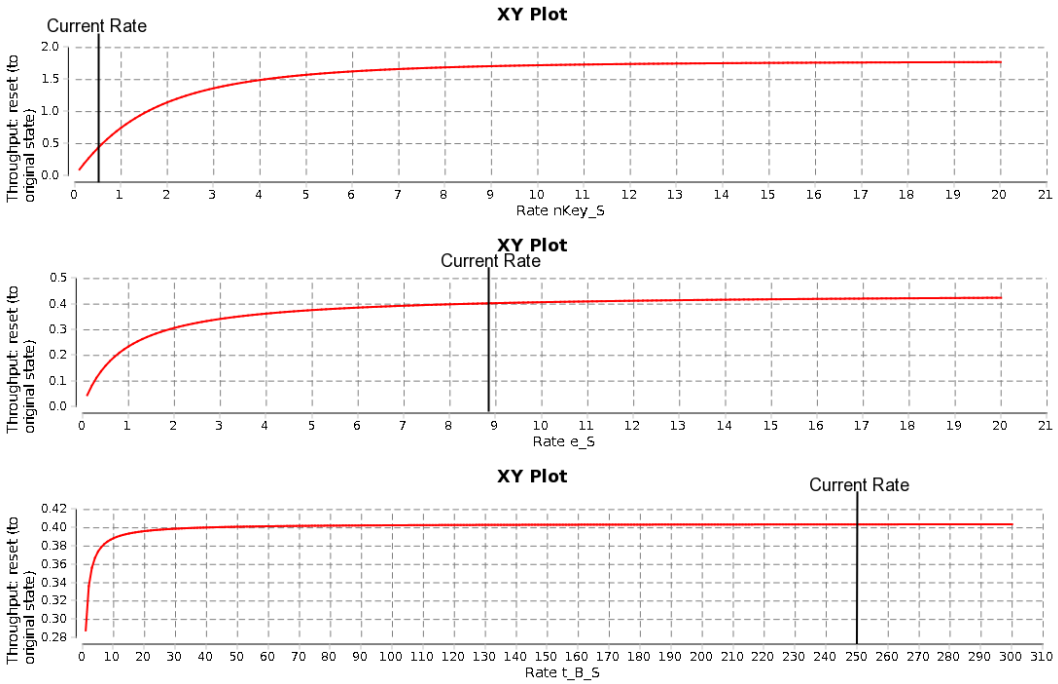


Figure 4.8: Experimentation Graph

or upgrading the hardware which will additionally slightly improve the rates of other operations for principal S.

4.6.3 Passage Time Analysis

The final analysis type we will be examining is passage time analysis. For our uses, this analysis allows us to provide a service level agreement or quality-of-service metric benchmarks with regard to the percentage chances of a protocol run successfully terminating within a set time period. Unlike the previous sections we added this particular analysis type to the existing PAM framework. The analysis itself is undertaken by a Java version of the HYDRA tool[17] called jHYDRA, both of which were developed at Imperial College London.

Due to the way we construct the PAM file we know that the source and destination states will be the first and last states in the file, this simplifies both the analysis and the user interface. The user must specify the start time and the end time as well as the step period. The tool then generates either a graph showing the probability density function or the cumulative distribution function. The probability density function graph shows the chance of the protocol reaching the target state, in other words terminating, at any

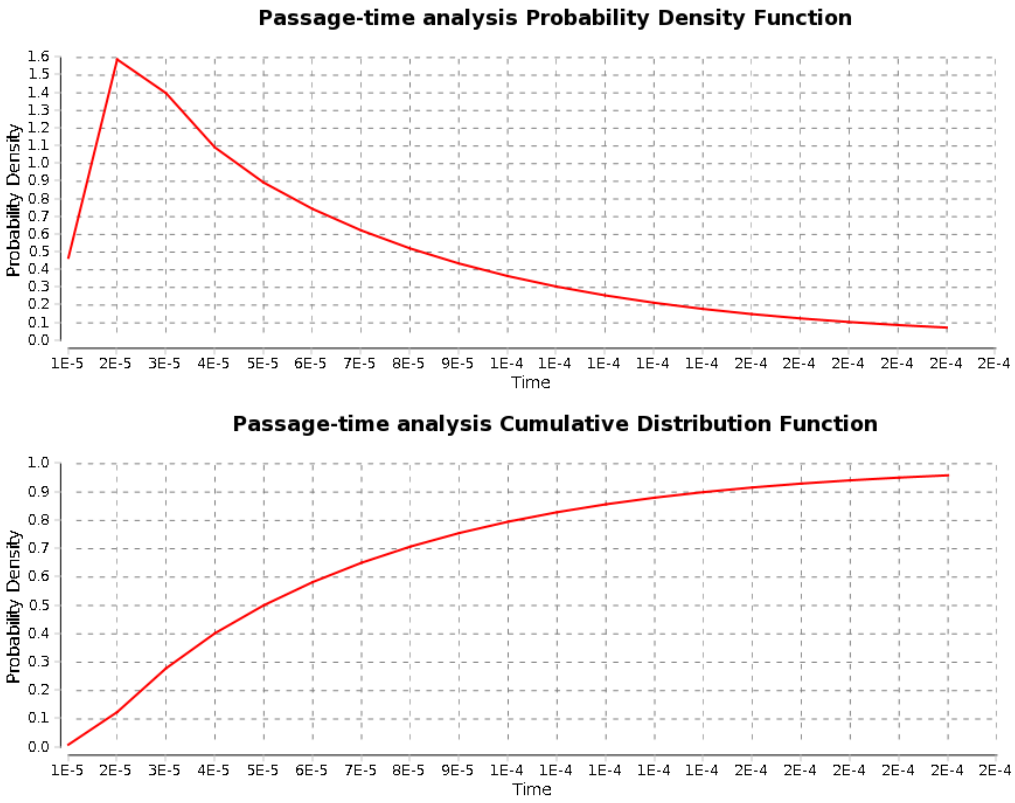


Figure 4.9: Passage Time Analysis Graphs

given point in the given time period. From this, it is possible to calculate the cumulative distribution. This allows us to state the probability that the protocol will terminate by a certain time.

To compensate for the AutoRate tool setting the reset rate as high as possible to reduce the time spent in the final target state, when generating the Passage Time Analysis we change it to an average of all other rates. As this rate is not used in the passage time analysis we can modify the rate without affecting the output and by using the average we reduce the amount of uniformisation needed by the jHYDRA tool.

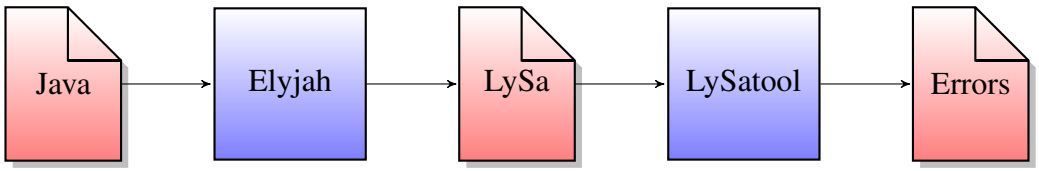
Figure 4.9 shows the two graphs created by passage time analysis inside the PAM framework. The top diagram is the probability density showing when the protocol is most likely to terminate. The second graph allows a reader to determine, for example, by what time is there a 90% probability the protocol will terminate. In this instance the value can be read off as about 0.018.

Chapter 5

Towards Formal Analysis of Implementations

5.1 Introduction

In this chapter we shall introduce the Elyjah tool which works to allow the analysis of implementations of cryptographic protocols in the Java programming language. This is accomplished by translating the Java program into LySa which is then analysed using the LySatool as described in Section 2.7.



The ability to analyse implementations as well as specifications allows users to verify that they have correctly implemented a cryptographic protocol. The Elyjah tool is designed to be used by those unfamiliar with the formal techniques used in this analysis but who are required to undertake protocol analysis in order to discharge professional responsibilities to deliver a secure product. It is assumed that those who already have experience working with designing protocols will prefer to use other methods of protocol analysis working directly with logical reasoning and semantic representations to analyse their new protocol ideas.

5.2 Requirements of the Elyjah Software

As we are developing a method for analysing Java implementations of protocols our tool has several design requirements particular to this scenario.

Naturally the first priority is for the model that Elyjah generates to be an accurate translation of the Java into LySa. The generated LySa should be formatted such that it is readable to a user but also able to be analysed by the LySatool without further editing required. In particular this means that we generate LySa models in the concrete syntax accepted by the LySatool, not the mathematical syntax used in scientific publications.

An important requirement is that the Java programs are fully runnable programs. This means the programs should be able to contain extra code that does not directly relate to the protocol. This additional code should be ignored by Elyjah when its presence does not alter the protocol model.

It is also important that our technique analyses the code rather than any comments or extraneous information in the source code file which the developers have added to explain the intended operation of the code. We want to know that the model generated is an accurate representation of the code.

As this tool is to be used as part of a development process it should run quickly enough that it can be used every time that the file is compiled without excessively delaying the developer. Additionally the tool should be presented in a way that makes it easy to use as part of the development process. Allowing developers to validate the correctness of their code as part of the development cycle reduces the need for continued updating and patching of software.

5.3 The Java-LySa Application Programming Interface

Although it would be possible for a software tool to be able to accept any Java program as input before converting the source code into a LySa process, it would be substantially less reliable than one which demands a uniform input format. Thus a framework is needed to allow developers to model protocols, while also allowing the software tool to parse the source code and identify the security-critical operations. We have designed and encapsulated an API which provides the required functionality for message composition, encryption and communication. This API is called the Java-LySa API, or JaLAPI for short. An implementation of a protocol needs to be a directly executable Java program which the developer can use to test that the protocol functions as ex-

pected before analysing its security properties. In order for it to be possible for any tool to understand the developer's intent, there needs to be a standard method of achieving certain goals. As such there needs to be a uniform approach to encrypting/decrypting data and a uniform method of sending messages between principals. JaLAPI gives a developer easy access to such cryptographic and communication functions and makes it easy for an automated tool to find and analyse these functions.

5.3.1 JaLAPI Requirements

Each principal must be modelled in a separate class which extends the `Thread` class thus allowing multiple principals to be running at the same time, thereby simulating LySa's concurrent processes. There will also need to be an additional class whose main method contains the code to set up the principals; create any keys known prior to the commencement of a protocol; and finally establish the network through which all principals will communicate. This network will also need to be implemented as a separate class which is capable of keeping references to the various principals. Finally, there needs to be a class for generating keys as both the principals and protocol initialisation class will need to be able to do this and should use the same implementation to do so.

There needs to be an abstract class set up which implements most of the functions of a principal while leaving un-implemented the `run` method (inherited from the `Thread` class) and a method to deal with an incoming message which will be called `processIncoming`. The `run` method will be used to start the protocol, for example, if principal *A* sends the first message in a protocol, then *A*'s `run` method will contain this code. Other functions of the class that are required would be: to organise a key store for the three possible kinds of key: `SecretKey`, `PublicKey` and `PrivateKey`; and to encrypt and decrypt blocks of messages, allowing for crypto-point annotation at these points. These methods will be used as keywords so that the parser can pick out the key parts of the protocol. As a developer is likely to want to be able to print out the values of variables and other debugging information, a `Logger` should be set up in the abstract class to encourage developers to use this rather than the less structured `println` system call. In addition, a uniform method of dealing with incoming messages is needed. Elyjah uses case/switch blocks to separate the code for dealing with individual messages. A variable stores which message is expected next and each time the `processIncoming` method is called this variable is incremented. Pattern matching is modelled by using a method `check` which validates that the value at each position

of the incoming messages matches the value which is expected. The network class only needs methods to send a message from one principal to another and some method of registering principals. The key generation class needs to have a method to return a `SecretKey` for symmetric encryption and another method to return a `KeyPair` for asymmetric encryption. Messages will be a custom object named `Message`. The `send` command will need to have an argument for this object and the intended recipient of the message.

Encryption and decryption methods will also work on the same bespoke message objects and will require optional arguments for crypto-points. This will require several different overloadings of these methods as it is possible for the crypto-point to specify only the current location, one origin/destination, multiple origins/destinations or neither.

5.3.2 The JaLAPI `KeyGeneration` class

Key generation is required for principals when in a protocol run but additionally it is needed for any keys which must be shared before a protocol is run. We need to generate keys for both symmetric and asymmetric encryption types. The `KeyGeneration` class contains two methods for generating different types of keys, `generateSharedKey` and `generateKeyPair`. The JavaDoc for these methods is presented in Table 5.1. These return a `SecretKey` and a `KeyPair` respectively. These are both objects provided by standard Java classes. A `SecretKey` is part of the `javax.crypto` package and the `KeyPair` is provided by java.security. The `KeyPair` object is a holder for a public and a private key which can be retrieved with `getPublic` and `getPrivate` method invocations. There is a hierarchy for Key objects which is represented by Figure 5.1. This allows us to use a key store for objects of the parent type `Key` which is sufficient for all sub-types.

5.3.3 The JaLAPI `Principal` class

This class provides all the functionality needed by a principal in a cryptographic protocol. In order to deal with incoming messages the class must override the abstract `processIncoming` method. To help with this we provide a `check` method which helps with the pattern matching section of the message processing.
The encryption-related methods are presented in Table 5.3. As we will be dealing with

Method Summary	
`static java.security.KeyPair`	`generateKeyPair(long userseed)` Returns a KeyPair object that can then be used to retrieve matching PublicKey and PrivateKey objects.
`static javax.crypto.SecretKey`	`generateSharedKey()` Returns a SecretKey object that can then be used to encrypt Message objects.

Table 5.1: The JaLAPI `KeyGeneration` class methods

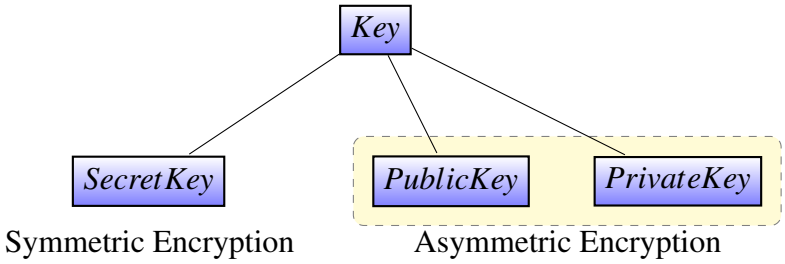


Figure 5.1: Key Hierarchy in Java

encryption and decryption on blocks of text we will use our bespoke `Message` object as a wrapper for plain-text message parts. The decryption and encryption methods are also overloaded with methods which have additional parameters representing crypto-points. These parameters are a `String` for the label of the current crypto-point and either another `String` representing a single origin or destination or an array containing multiple possible crypto-points. These overloaded methods discard the additional parameters and call the standard `encrypt` or `decrypt` methods.

Table 5.4 lists the methods which are used to represent the creation of ν LySa processes. While `generateMessage` is used as a signpost without doing any processing and simply returns the input parameter, `generateNonce` does produce a String representation of a nonce. Methods for generating a `SecretKey` or a `KeyPair` call the same methods in the `KeyGeneration` class.

We provide a key-store so that we can store keys along with a label which will be used to refer to the key in the LySa model. This allows us to consistently use the same name to refer to the key across different principals without worrying about the name of the variable that the developer used in each instance. The methods detailed in Table 5.5

Method Summary	
`abstract` void	`processIncoming(Message msg)`
	Override this to process this incoming message
`void`	`check(java.lang.Object a, java.lang.Object b)`
	Compare the two parameters, throws an error if they are not equal

Table 5.2: The JaLAPI `Principal` class methods for incoming messages

Method Summary	
`Message`	`decrypt(java.lang.String str, java.security.Key key)`
	Decrypts an encrypted message with given key
`java.lang.String`	`encrypt(Message msg, java.security.Key key)`
	Encrypts a Message with given key
`java.lang.String`	`hash(Message msg)`
	Hash function

Table 5.3: The JaLAPI `Principal` class methods for encryption and decryption

Method Summary	
`java.lang.String`	`generateMessage(java.lang.String message)`
`java.lang.String`	`generateNonce()`
`javax.crypto.SecretKey`	`generateSharedKey()`
	Returns a SecretKey object that can then be used to encrypt Message objects.
`java.security.KeyPair`	`generateKeyPair(long userseed)`
	Returns a KeyPair object that can then be used to retrieve matching PublicKey and PrivateKey objects.

Table 5.4: The JaLAPI `Principal` class methods for generating new objects important to a cryptographic protocol

allow a developer to store a key in the key store along with this label and retrieve it at a later point in the protocol run.

The `sendKey` method is used to return a `Key` in a transmissible format that can

Method Summary	
void	registerKey(java.security.Key key, String name) Register key in keystore with accompanying label
java.security.Key	getKey(String name) Retrieve a Key from the keystore which was stored with the given label

Table 5.5: The JaLAPI `Principal` class key-store related methods

be sent in a `Message` and then reformatted into the appropriate `Key` type. We provide several methods which take a serialised version of a `Key` and turn it into the appropriate format.

Method Summary	
java.lang.String	sendKey(java.security.Key key) Returns String representation of Key
java.security.PrivateKey	receivePrivateKey(java.lang.String str) Turns a String representation of a private key into a PrivateKey object
java.security.PublicKey	receivePublicKey(java.lang.String str) Turns a String representation of a public key into a PublicKey object
javax.crypto.SecretKey	receiveSecretKey(java.lang.String str) Turns a String representation of a symmetric key into a SecretKey

Table 5.6: The JaLAPI `Principal` class methods for sending and receiving `Key` objects

5.3.4 The JaLAPI `Network` class

The network class is primarily used to pass messages between principals. We register principals with a network then send messages using the name we register them with. As well as passing the message to the correct principal, the `Network` layer also sets the source and destination of the message prior to transmission. It is also used to pass any keys that need to be known by multiple principals before a protocol begins.

Method Summary	
void	register(java.lang.String name, Principal comClass)
	Register new principal with the network
void	send(java.lang.String dest, Message msg)
	Send a message to a principal that has registered with the network
void	shareKey(java.security.Key key, java.lang.String name)
	Send a key to all principals

Table 5.7: The JaLAPI `Network` class methods

5.3.5 The JaLAPI `Message` class

In order to be able to exercise control over the manipulation of message contents we provide an implementation of a message. By providing a bespoke object with additional fields for the source and destination rather than using a standard variation of a `List` we restrict a developer's options to modify the contents. With this we can control the order that message parts are added and inspected making the analysis more secure. The following public methods are provided:

Constructor Summary	
Message()	
	Creates a new empty Message object

Method Summary	
void	add(java.lang.String msgPart)
	Adds a new String to the message
java.lang.String	getDest()
	Returns String representation of intended message destination.
java.lang.String	getNext()
	Returns the next part of the message.
java.lang.String	getSource()
	Returns String representation of message source.
java.lang.String	toString()
	String representation of message contents

Table 5.8: The JaLAPI `Message` class methods

We also provide methods for setting the source and destination of the message but access to these methods is restricted to other classes in the JaLAPI package. This is so that they are set by the `Network` class and cannot be modified by the user.

5.3.6 JaLAPI Trace

In the provided JaLAPI implementation, certain methods produce logging information. This serves two purposes that are both related to the tools presented in Chapters 3 and 4.

The first usage is to generate protocol narration similar to that provided by the AnaLySa as described in Section 3.3. This execution trace shows the message sent but leaves out encrypted message block contents providing only the number of parts.

The second function is that for tasks such as encryption or message part generation the time taken for these actions to complete is provided to the user to use as a rate for the AutoRate tool as defined in Section 4.5.

5.4 Elyjah Implementation

Elyjah identifies method calls to the provided API. Using these method calls as signposts, the cryptographic and communication parts of the security protocol can be divined. This is required as Java is more expressive than LySa so the language needs to be restricted in order to be able to achieve accurate translation. This means that Elyjah allows a Java implementation of a cryptographic protocol to be translated into a formal model. The LySa model can be analysed using the LySatool to return the security properties of the protocol implementation. While Elyjah cannot analyse the source code of an arbitrary communications package, it is possible to implement a protocol that can be translated by Elyjah and used in a full application.

5.4.1 Java Parser and Abstract Syntax Tree

Before working on the conversion process itself, it is necessary to create a Java parser. Like the LySa parser from Section 3.1 we use the combination of JJTree and JavaCC to produce an abstract syntax tree similar to the one for LySa. It is then possible to navigate the abstract syntax tree much as any other tree. Classes, methods and even single lines of code have a tree structure so a single `SimpleNode` can be used as a pointer to any part of the code. An example of one line of code represented as an abstract syntax tree is given in Figure 5.2.

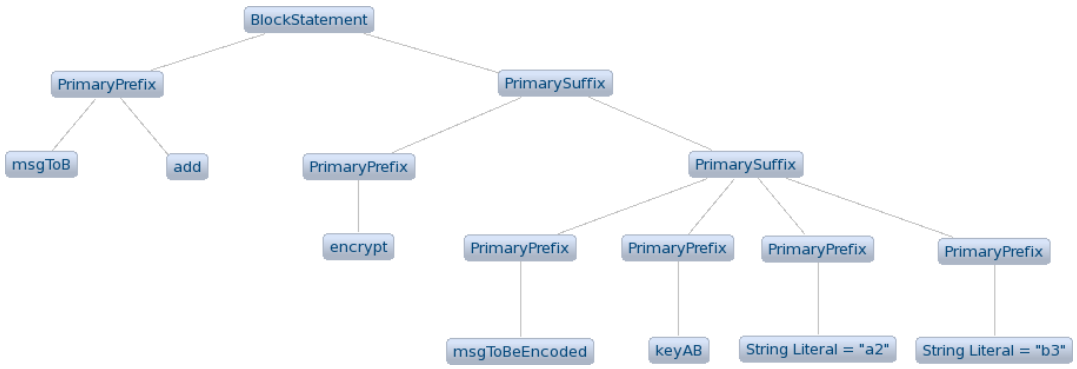


Figure 5.2: Syntax Tree of msgToB.add(encrypt(msgToBEncrypted, keyAB, “a2”, “b3”));

5.4.2 Code Analysis

The first stage in analysing a Java application is to scan the complete Java source file to locate the roots of the separate principal classes. A class name can be found by searching the syntax tree for a node containing the string ClassOrInterfaceDeclaration. Once this node is found the name of the class is located in the first child of this node.

It is also necessary to have a mapping between the name of classes and the name used to register them with the `Network` class. By scanning the set-up class we find the invocations of the network's `register` method in order to identify the name of the principals. This set up class is also scanned for any method invocations of the `KeyGeneration` class which will be used to generate some LySa code. This process will be repeated later so we place this in a separate method so that we can re-use it later. Once this set-up class is scanned we analyse each of the LySa classes in turn.

For each class we identify the name of the `Network` object by analysing the source code and examining the name of the variable where we save the `Network` in the constructor. We then determine whether the class contains a run method. If it does we start our analysis of the class here by firstly generating new expressions (Section 5.4.2.1) and then generate any send expressions present in this block of code (Section 5.4.2.3). After this Elyjah generates the LySa code equivalent to the Java code in the principal's `processIncomingMessage` method as described in Section 5.4.2.6.

5.4.2.1 Generating New Expressions

This method is applied to a block of code, and searches for all Java code which is equivalent to LySa processes involving the ν operator. We search recursively down from the parent searching for certain method calls. If we find invocations of the meth-

ods `generateMessage` or `generateNonce` then we update our LySa code with the name of the variable in a LySa restriction process such as:

(new variable)

If the method is `generateSharedKey` we firstly find the name of the variable used to store the `SecretKey`. However in the LySa model we want to refer to the name of the key and not the variable in which it happens to be stored. For this reason we search the surrounding block of code to identify the label using the technique described in Section 5.4.2.2. The returned name of the key is then used in our LySa code in

(new keyLabel)

If an invocation of the `generateKeyPair` method is found then we get the variable name from the method invocation and resolve the name of the key pair by finding the name of the variable of either a public or private key generated by the key pair by searching for method invocations `getPrivate` or `getPublic`. We use the process in Section 5.4.2.2 to get the label this key is registered with and if necessary remove either the + or - symbol to obtain a general name for the pair of keys.

5.4.2.2 Resolving Keys

We firstly identify the key type based on the type of variable used. This will either be `SecretKey`, `PublicKey` or `PrivateKey`. Then we search for calls to methods accessing or modifying the key store to find the label the key is stored with. These will typically be either the methods for registering the key or getting the key defined in Table 5.5 but could also be the `shareKey` method used in the set up class to initialise keys that were set up before the protocol starts. If dealing with asymmetric encryption then we append a + or - if the key is a public or private key and does not have a sign at the moment.

5.4.2.3 Generating Send Processes

We already know the `Network` name and using this we search for all method invocations of this object's send method. For each of these that we find, we identify the destination from the method invocation. The LySa send process begins with the first two tuples being the source and intended destination of the message. We can also identify the name of the `Message` from the method invocation. We then determine the contents of the message.

5.4.2.4 Processing Message Contents

We first find all `messageName.add` method invocations. For each of these we see whether what we are adding contains a method invocation to `encrypt` or `hash`, in that case we determine the contents of the plain-text with the techniques decribed in Section 5.4.2.5 and append the returned LySa code from this method. If there is a method invocation of `sendKey` then we resolve the label of this key as in Section 5.4.2.2 and we append this to the contents of the message. If we are adding a string literal we append this and if we are appending a variable we determine if this is an encrypted block and if so we determine the contents.

5.4.2.5 Generating Encrypted Blocks

We append the opening to a LySa encryption portion, and if the encryption is asymmetrical we denote this syntactically. By examining the `encrypt` or `hash` method invocation we determine the name of the `Message` object. We append the contents of the encrypted portion by repeating the process in the previous section on this `Message` object. We close the encrypted portion and specify the encryption key. If we are hashing rather than encrypting then we use the key is "MD5+" to denote public key encryption with specifying a corresponding key which can be used to decrypt the message. If this is a normal encryption then we get the variable name from the method invocation and use the technique in Section 5.4.2.2 to get the correct key name. Crypto-points are then attempted to be added from the method invocation with exception handling to cope with any absences. If the destination crypto-point is an array of possible crypto-points then the same technique as identifying the contents of a `Message` presented in Section 5.4.2.4 is used.

5.4.2.6 Analysing a Principal's `processIncomingMethod`

The first task is to identify the name of the `Message` object containing the incoming message from the parameter in the name of the method header. We then separate the method into different switch statements. We tackle each of these individually and in order generate LySa restriction processes, input processes followed by decrypt and finally send processes.

5.4.2.7 Generating Input Processes

Initially, the task is to find all method invocations of the `Message` object's `check` method. This includes `checkSource` and `checkDest` which are used to form the first two parts of the tuple. These two followed by the other checks are appended to the LySa model. We insert a separator and then continue finding all method invocations of the `getNext` method of the `Message`.

5.4.2.8 Generating decrypt Processes

Firstly find decrypt statements of the form

```
Message d = decrypt(stringName, keyVariable, crypto1, crypto2)
```

The start of the LySa process for decryption can then be appended

decrypt stringName as{

specifying asymmetric decryption as needed. In order to process the contents of the encrypted message we proceed as in Section 5.4.2.7. Namely we first process the invocations of the `check` to the `Message` referred to in the Java `decrypt` method invocation. As before we continue finding all method invocations of the `getNext` method of the `Message` then determine the key name by using the process in Section 5.4.2.2 on the variable `keyVariable`. Crypto-points are dealt with as in encryption.

5.5 Eclipse IDE Integration

This whole process works in the Eclipse IDE, and is designed to immediately give the developer feedback as to the security properties of their protocol implementation. Elyjah is provided as an Eclipse plug-in and is available as a compilation-time tool to analyse a Java file and provide feedback in the same way as the Java compiler might, by opening a console in the IDE. Figure 5.3 shows Elyjah running as an Eclipse plug-in producing the LySa model. The results of the LySatool are also presented to the developer in the Elyjah console, giving them immediate feedback on the security properties of their implementation.

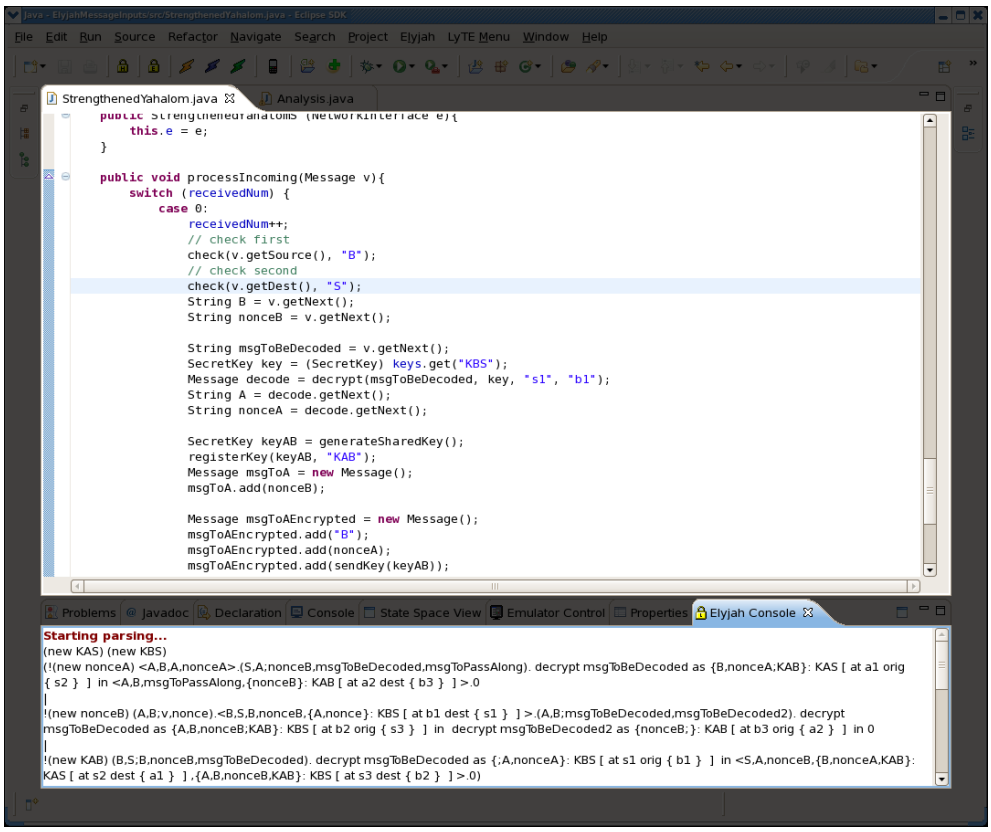


Figure 5.3: Screenshot of Eclipse IDE with Java editor and Elyjah console highlighted

5.5.1 Performance

The table below shows performance figures for a range of cryptographic protocols. These figures demonstrate that these tools are efficient enough to be used as part of a usual development cycle. Equally, it is worth noting that these times are much lower than those exhibited by tools such as FS2PV[9] or the much more efficient follow up F7[7]. Comparison of performance timing with such tools is hard as the inputs are vastly different and the programs are running on different computers. Despite this, it is worth noting that FS2PV takes eight and a half minutes to analyse an implementation of the Otway-Rees protocol and F7 takes a minute and a half to do so.

Protocol	Implementation			Timings	
	LoC	Messages	Principals	Elyjah	LySatool
Needham-Schroder	238	6	3	0.055s	0.156s
Otway-Rees	229	5	3	0.063s	0.143s
Wide Mouth Frog	157	3	3	0.042s	0.126s
Yahalom	219	4	3	0.063s	0.147s

5.6 Worked Examples

5.6.1 Naive Public Key Protocol Example

In order to demonstrate this process, we will present the implementation and analysis of a very simple public key protocol. The standard narration is as follows:

$$A \rightarrow B : K+$$
$$B \rightarrow A : \{msg\}_{K+}$$

The clear goal of this protocol is to allow secure communication between principals *A* and *B* by having *B*'s message to *A* being encrypted. By implementing this model in Java and then using Elyjah and the LySatool we will prove that this goal is not achieved.

The following excerpts are taken from a single Java file which simulates the execution of the above protocol. It is important to note that it represents a runnable program that can also be run over a distributed environment. The two principals are implemented in separate classes, which run as separate threads, within one Java file. These principals must extend the abstract class named `Principal`. This abstract class implements key management and encryption functionality and provides an abstract `processIncoming` method for developers to extend to implement receiving messages. The `Principal` class itself extends the `Thread` class allowing the principal to be run concurrently with other principals in the protocol.

Firstly the principals and network must be set up as below. Any keys shared between principals must also be generated and shared at this stage. Although this is not required in this example, it can be seen in Figure 5.3.

```
public class PKEncryptionExample extends KeyGenerationClass{
    public static void main(String[] args) {
        Network net = new Network();
        PKEA a = new PKEA(net);
        net.register("A", a);
        PKEB b = new PKEB(net);
        net.register("B", b);

        a.start();
        b.start();
    }
}
```

5.6.1.1 Principal A's first message

In order to initiate the protocol the following code is contained in the principal's `run` method.

```
public void run(){
    KeyPair keypair = generateKeyPair(1024);
    PrivateKey privateKey = keypair.getPrivate();
    PublicKey publicKey = keypair.getPublic();
    registerKey(privateKey, "K-");
    registerKey(publicKey, "K+");
    Message v = new Message();
    v.add(sendKey(publicKey));
    net.send("B", v);
}
```

The first three lines generate asymmetric keys using a call to the provided API's `generateKeyPair` method which returns a `KeyPair` object created by a series of calls to Java's built-in cryptography library. The calls to the `registerKey` method have two purposes. In the context of the program it allows the keys to be stored and used again in other methods. The additional function allows Elyjah to map the variable name used for the key to the name given. The final three lines initialise, populate and send the message from *A* to *B* using the provided bespoke Message class and Network class represented by the object `net`. The `sendKey` method is used to convert the `PublicKey` object into a `String` object for transmission. Due to LySa using the key name defined in the `registerKey` method call the LySa code will appear not as `publicKey` but K+.

5.6.1.2 Principal B's receipt of message and reply

This code is from principal B's `processIncoming` method.

```
public void processIncoming(Message v){
  switch (receivedNum) {
    check(v.getSource(), "A");
    check(v.getDest(), "B");
    PublicKey key = receivePublicKey(v.getNext());
    registerKey(key, "K");
    Message vEnc = new Message();
    String message = generateMessage("secret message");
    vEnc.add(message);
```

```
        Message v2 = new Message();
        v2.add(encrypt(vEnc, key, "b1", "a1"));
        net.send("A", v2);
    }
}
```

The first four lines deal with the incoming message from *A*. The first two lines check that the source and destination match the expected participant's roles. The following two lines firstly use the `receivePublicKey` method to transform the `String` object into a `PublicKey` which is then registered with the principal's keystore. The next block of code deals with the construction of *B*'s reply. Encrypted parts of the message are represented by another `Message` object. Encryption and adding the subset to the message occur at the same time in the penultimate line of the code. The `encrypt` method call returns a `String` object which is an encrypted representation of the `Message` contents. The final two parameters of the `encrypt` method call represent the crypto-points that the LySatool uses to analyse security properties. The first parameter is a label for "at" and the second defines where this message should be decrypted.

5.6.1.3 Principal A's decryption of message

This code is from principal *A*'s `processIncoming` method.

```
public void processIncoming(Message v){
    switch (receivedNum) {
        check(v.getSource(), "B");
        check(v.getDest(), "A");
        String msgToBeDecoded = v.getNext();
        PrivateKey keyD = (PrivateKey) getKey("K–");
        Message decode =
          decrypt(msgToBeDecoded, keyD, "a1", "b1");
        String msg = decode.getNext();
        theLogger.info(msg);
    }
}
```

This block highlights the decryption process, this involves retrieving a key from the principal's keystore and converting a `String` into a `Message` object. This new `Message` can then have parts checked or assigned to variables like any other `Message`.

5.6.1.4 LySa model from Elyjah

The result of running Elyjah on the Java file representing this protocol is given below. The first two lines represent principal *A*, the last line represents principal *B*.

(ν ± K)⟨A,B,K+⟩.(B,A;msgToBeDecoded).
decrypt msgToBeDecoded as $\{\|;msg\|\}_{K-}$ [at a1 orig { b1 }] in 0
|
(ν message) (A,B;K).⟨B,A,$\{\|message\|\}_K$ [at b1 dest { a1 }] ⟩.0

Table 5.9: LySa model of Naive Public Key Encryption

5.6.1.5 LySatool result and analysis

This analysis by the LySatool reveals several attacks. The first section reveals the message parts that can be read by an attacker. The worrying item here is that the message which is intended to be secret is included here. This is because an attacker can pose as *A*, start a conversation with *B* with the attacker's own public key and decrypt it with their own private key. Equally an attacker can intercept *A*'s message to *B* and reply with their own message, posing as *B*, clearly a problem if *A* interprets this message as a command.

Values that may not be confidential K+, $\{\|Lmessage\|\}_{l\bullet}$ [at b1 dest a1], n•, m•+, m•-, B, A, $\{\|l\bullet\|\}_{l\bullet}$ [at CPDY], message
Violation of authentication properties (Ψ) (b1, CPDY), (CPDY, a1)

5.6.2 Otway-Rees Protocol

As a more complete example we also take a look at the Otway-Rees key sharing protocol[61]. This protocol establishes a fresh symmetric key for communication between two principals *A* and *B*. This protocol relies on the presence of a trusted server *S* who already has long term keys *KAS* and *KAB* for communicating with principals *A* and *B* respectively. As presented in the original text the protocol is as follows:

1. $A \rightarrow B : M, A, B, \{N_A, M, A\ , B\}_{K_A}$
2. $B \rightarrow S : M, A, B, \{N_A, M, A\ , B\}_{K_A}, \{N_B, M, A\ , B\}_{K_B}$
3. $S \rightarrow B : M, \{N_A, K\}_{K_A}, \{N_B, K\}_{K_B}$
4. $B \rightarrow A : M, \{N_A, K\}_{K_A}$

As mentioned in Section 2.5.2 some protocols require the message parts to be reorganised in order to encode the protocol in LySa. The encrypted sections in messages 1 and 2 need to be rearranged to place the nonces N_A and N_B at the end of the encrypted section so that we can bind these to variables and check the other message parts. An additional final step has been added to the protocol where principal B sends a secret message to principal A using the fresh key. This allows the LySatool to validate that secure communication using this key is possible. This updated protocol is shown below.

1. $A \rightarrow B : M, A, B, \{M, A\ , B, N_A\}_{K_A}$
2. $B \rightarrow S : M, A, B, \{M, A\ , B, N_A\}_{K_A}, \{M, A\ , B, N_A\}_{K_B}$
3. $S \rightarrow B : M, \{N_A, K\}_{K_A}, \{N_B, K\}_{K_B}$
4. $B \rightarrow A : M, \{N_A, K\}_{K_A}$
5. $B \rightarrow A : \{msg\}_K$

This protocol differs from the one in Section 5.6.1 in several ways, not least that keys are shared beforehand. This peculiarity is shown in the following section of Java code, the set up class.

```
public static void main(String[] args) {
    Network net = new Network();
    ORA a = new ORA(net);
    net.register("A", a);
    ORB b = new ORB(net);
    net.register("B", b);
    ORS s = new ORS(net);
    net.register("S", s);

    SecretKey keyA = generateSharedKey();
    SecretKey keyB = generateSharedKey();

    a.shareKey(keyA, "KAS");
    s.shareKey(keyA, "KAS");
    b.shareKey(keyB, "KBS");
```

```
        s.shareKey(keyB, "KBS");

        a.start();
        b.start();
        s.start();
    }
```

5.6.2.1 LySa model from Elyjah

The result of running Elyjah on the Java file representing this protocol is given below. The first two lines represent the situation before the protocol begins with keys shared between the server and each of the principals. Technically, this model represents that all protocols know both these keys but looking through the rest of the protocol it is clear that only principals *A* and *S* know the key *KAS* and equally only *B* and *S* know *KBS*. The next three lines represent principal *A*, the middle three represent principal *B* and the final three lines represent principal *S*.

Analysis of this model indicates that no crypto-point assertions are broken and that the

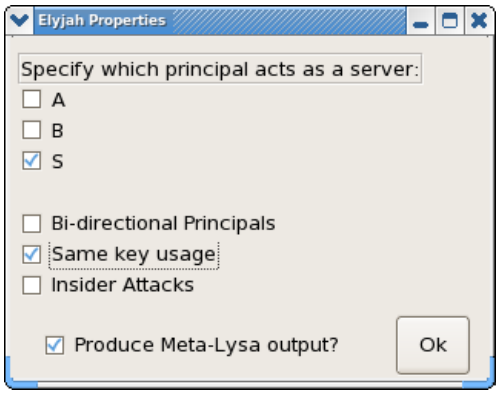
(ν KAS) (ν KBS)
(!(ν NA) ⟨A,B,M,A,B,$\{$A,B,M,NA$\}_{KAS}$ [at a1 dest { s1 }] ⟩.(B,A,M;x1). decrypt x1 as $\{$NA;xk$\}_{KAS}$ [at a2 orig { s3 }] in (B,A;x2). decrypt x2 as $\{$;xmsg$\}_{xk}$ [at a3 orig { b3 }] in 0
|
!(ν NB) (A,B,M,A,B;y1).⟨B,S,M,A,B,y1,$\{$A,B,M,NB$\}_{KBS}$ [at b1 dest { s2 }] ⟩.(ν MSG) (S,B,M;y2,y3). decrypt y3 as $\{$NB;yk$\}_{KBS}$ [at b2 orig { s4 }] in ⟨B,A,M,y2⟩. ⟨B,A,$\{$MSG$\}_{yk}$ [at b3 dest { a3 }] ⟩.0
|
!(ν K) (B,S,M,A,B;z1,z2). decrypt z1 as $\{$A,B,M;zna$\}_{KAS}$ [at s1 orig { a1 }] in decrypt z2 as $\{$A,B,M;znb$\}_{KBS}$ [at s2 orig { b1 }] in ⟨S,B,M,$\{$zna,K$\}_{KAS}$ [at s3 dest { a2 }] ,$\{$znb,K$\}_{KBS}$ [at s4 dest { b2 }] ⟩.0)

Table 5.10: LySa model of Otway-Rees Encryption

supposedly secret *msg* sent in the final message is indeed confidential and an attacker cannot read it.

5.7 Meta-LySa and the Deployment Scenario Solution

Introduced in Section 2.5.4, Meta-Level LySa allows us to describe the deployment scenario of a protocol. Protocols will often reveal different flaws when multiple copies of principals are running simultaneously. This situation allows an attacker to use messages from one session in another session either parallel or at a later time. The LySa model in Table 5.10 represents a situation where there is a single principal A and a single principal B who initiate a single session with a server S. In the rest of this section we will use Elyjah to generate more general deployment scenarios and see if the LySatool analysis changes with these models. When generating LySa models Elyjah offers users the chance to add meta-level indicies through the following pop-up dialogue.



5.7.1 Multiple Participants

The LySa model in Table 5.11 describes the most basic parallel situation where a principal S acts as a solo server and there exists an arbitrarily large set of initiators who attempt to establish communication with each acting as either initiators or responders. Principals share a pair of keys with the server. One for when they act as an initiator and another for when they are assuming the role of a responder. In this scenario, a principal will use the key KAS when acting as a initiator and the key KBS as a responder. Analysing this example with the LySatool presents the same results as with the standard LySa model in Table 5.10. This tells us that with distinct roles for principals and with principals using different keys for these different roles adding multiple versions of principals A and B does not negatively affect the security properties of the protocol.

let R $\subseteq \mathbb{N}$ in
let S $\subseteq \mathbb{N}$ in
($\nu_{i\in R}$ KAS$_i$)($\nu_{j\in S}$ KBS$_j$)((|$_{i\in R}$ |$_{j\in S}$!(ν NA$_{i,j}$) $\langle$A$_i$,B$_j$,M$_{i,j}$,A$_i$,B$_j$,{A$_i$,B$_j$,M$_{i,j}$,NA$_{i,j}$ } KAS$_i$ [at a1$_{i,j}$ dest { s1$_{i,j}$ }] $\rangle$.(B$_j$,A$_i$,M$_{i,j}$;x1$_{i,j}$). decrypt x1$_{i,j}$ as {NA$_{i,j}$;xk$_{i,j}$ } KAS$_i$ [at a2$_{i,j}$ orig { s3$_{i,j}$ }] in (B$_j$,A$_i$;x2$_{i,j}$). decrypt x2$_{i,j}$ as {;xmsg$_{i,j}$ } xk$_{i,j}$ [at a3$_{i,j}$ orig { b3$_{i,j}$ }] in 0)
|
(|$_{i\in R}$ |$_{j\in S}$!(ν NB$_{i,j}$) (A$_i$,B$_j$,M$_{i,j}$,A$_i$,B$_j$;y1$_{i,j}$).$\langle$B$_j$,S,M$_{i,j}$,A$_i$,B$_j$,y1$_{i,j}$,{A$_i$,B$_j$,M$_{i,j}$,NB$_{i,j}$ } KBS$_j$ [at b1$_{i,j}$ dest { s2$_{i,j}$ }] $\rangle$.(ν MSG$_{i,j}$) (S,B$_j$,M$_{i,j}$;y2$_{i,j}$,y3$_{i,j}$). decrypt y3$_{i,j}$ as {NB$_{i,j}$;yk$_{i,j}$ } KBS$_j$ [at b2$_{i,j}$ orig { s4$_{i,j}$ }] in $\langle$B$_j$,A$_i$,M$_{i,j}$,y2$_{i,j}$ $\rangle$.$\langle$B$_j$,A$_i$,{MSG$_{i,j}$ } yk$_{i,j}$ [at b3$_{i,j}$ dest { a3$_{i,j}$ }] $\rangle$.0)
|
(|$_{i\in R}$ |$_{j\in S}$!(ν K$_{i,j}$) (B$_j$,S,M$_{i,j}$,A$_i$,B$_j$;z1$_{i,j}$,z2$_{i,j}$). decrypt z1$_{i,j}$ as {A$_i$,B$_j$,M$_{i,j}$;zna$_{i,j}$ } KAS$_i$ [at s1$_{i,j}$ orig { a1$_{i,j}$ }] in decrypt z2$_{i,j}$ as {A$_i$,B$_j$,M$_{i,j}$;znb$_{i,j}$ } KBS$_j$ [at s2$_{i,j}$ orig { b1$_{i,j}$ }] in $\langle$S,B$_j$,M$_{i,j}$,{zna$_{i,j}$,K$_{i,j}$ } KAS$_i$ [at s3$_{i,j}$ dest { a2$_{i,j}$ }] ,{znb$_{i,j}$,K$_{i,j}$ } KBS$_j$ [at s4$_{i,j}$ dest { b2$_{i,j}$ }] $\rangle$.0))

Table 5.11: Meta-LySa model of Otway-Rees Encryption

5.7.2 Bi-directional Key Establishment

This scenario could be more generalised by declaring that each principal can act as both an initiator and a responder in a protocol. In this situation we replace principals *A* and *B* with a new indexed principal *I*. In this situation every principal I_i will initiate a session with every other principal I_j while also responding to sessions started by principals I_j. Table 5.12 details a deployment scenario where principals use the same address for initiating and responding although use different keys for these different roles. Analysis of this deployment scenario does not alter the security properties of this protocol.

Another potential scenario would be where we use the same key when a principal is cast as either initiator or responder but use different addresses for these roles.

The analysis of this deployment scenario yields several errors. In fact we can detect errors in several of LySatool's output mechanisms. We have violations of crypto-points, supposedly secret messages not being confidential and variables being bound to items they should not be. The full list of crypto-point violations reported by the LySatool are presented with indices added and thus would take up several pages of this thesis,

let R $\subseteq \mathbb{N}$ in
let S $\subseteq \mathbb{N}$ in
($\nu_{i\in R}$ KAS_i)($\nu_{j\in S}$ KBS_j)((|$_{i\in R}$ |$_{j\in S}$!(ν $NA_{i,j}$) $\langle I_i,I_j,M_{i,j}$,$I_i,I_j,\{I_i,I_j,M_{i,j}$,$NA_{i,j}$ $\}$ KAS_i [at $a1_{i,j}$ dest { $s1_{i,j}$ }] $\rangle$.($I_j,I_i,M_{i,j}$;$x1_{i,j}$). decrypt $x1_{i,j}$ as $\{NA_{i,j}$;$xk_{i,j}$ $\}$ KAS_i [at $a2_{i,j}$ orig { $s3_{i,j}$ }] in (I_j,I_i;$x2_{i,j}$). decrypt $x2_{i,j}$ as {;$xmsg_{i,j}$ } $xk_{i,j}$ [at $a3_{i,j}$ orig { $b3_{i,j}$ }] in 0)
|
(|$_{i\in R}$ |$_{j\in S}$!(ν $NB_{i,j}$) ($I_i,I_j,M_{i,j}$,I_i,I_j;$y1_{i,j}$).$\langle I_j,S,M_{i,j}$,$I_i,I_j,y1_{i,j}$,$\{I_i,I_j,M_{i,j}$,$NB_{i,j}$ $\}$ KBS_j [at $b1_{i,j}$ dest { $s2_{i,j}$ }] $\rangle$.(ν $MSG_{i,j}$) ($S,I_j,M_{i,j}$;$y2_{i,j}$,$y3_{i,j}$). decrypt $y3_{i,j}$ as $\{NB_{i,j}$;$yk_{i,j}$ $\}$ KBS_j [at $b2_{i,j}$ orig { $s4_{i,j}$ }] in $\langle I_j,I_i,M_{i,j}$,$y2_{i,j}$ $\rangle$.$\langle I_j,I_i,\{MSG_{i,j}$ $\}$ $yk_{i,j}$ [at $b3_{i,j}$ dest { $a3_{i,j}$ }] $\rangle$.0)
|
(|$_{i\in R}$ |$_{j\in S}$!(ν $K_{i,j}$) ($I_j,S,M_{i,j}$,I_i,I_j;$z1_{i,j}$,$z2_{i,j}$). decrypt $z1_{i,j}$ as $\{I_i,I_j,M_{i,j}$;$zna_{i,j}$ $\}$ KAS_i [at $s1_{i,j}$ orig { $a1_{i,j}$ }] in decrypt $z2_{i,j}$ as $\{I_i,I_j,M_{i,j}$;$znb_{i,j}$ $\}$ KBS_j [at $s2_{i,j}$ orig { $b1_{i,j}$ }] in $\langle S,I_j,M_{i,j}$,{$zna_{i,j}$,$K_{i,j}$ } KAS_i [at $s3_{i,j}$ dest { $a2_{i,j}$ }] ,{$znb_{i,j}$,$K_{i,j}$ } KBS_j [at $s4_{i,j}$ dest { $b2_{i,j}$ }] $\rangle$.0))

Table 5.12: Meta-LySa model of Otway-Rees Encryption with Bi-Directional Key Establishment

presented below are these violations with indices and repetitions removed.

Violation of authentication properties (ψ)
(a1, CPDY), (a1, s2), (b1, CPDY), (b1, s1), (b3, CPDY), (s3, b2), (s3, CPDY), (s4, a2), (s4, CPDY), (CPDY, a2), (CPDY, a3), (CPDY, b2), (CPDY, s1), (CPDY, s2)

The analysis also reports that the following crypto-point assertions do not hold within a single session (b3, a3), (s3, a2), (s4, b2).

The analysis also tells us that variables are bound to incorrect values.

$$\rho(zna_{i,j}) = NB_{i,j}, NA_{i,j}$$
$$\rho(znb_{i,j}) = NB_{i,j}, NA_{i,j}$$

Variable $zna_{i,j}$ should only ever be bound to variable $NA_{i,j}$ and equally variable $znb_{i,j}$ is only intended to be bound to variable $NB_{i,j}$. These attacks correspond to type flaw attacks where a principal is tricked into establishing a connection with itself.

let R $\subseteq \mathbb{N}$ in
let S $\subseteq \mathbb{N}$ in
($\nu_{i \in R}$ KS_i)((|$_{i \in R}$ |$_{j \in S}$!(ν $NA_{i,j}$) $\langle I_i,I_j,M_{i,j}$,$I_i,I_j,\{I_i,I_j,M_{i,j}$,$NA_{i,j}$ } KS_i [at $a1_{i,j}$ dest { $s1_{i,j}$ }] $\rangle$.($I_j,I_i,M_{i,j}$;$x1_{i,j}$). decrypt $x1_{i,j}$ as {$NA_{i,j}$;$xk_{i,j}$ } KS_i [at $a2_{i,j}$ orig { $s3_{i,j}$ }] in (I_j,I_i;$x2_{i,j}$). decrypt $x2_{i,j}$ as {;$xmsg_{i,j}$ } $xk_{i,j}$ [at $a3_{i,j}$ orig { $b3_{i,j}$ }] in 0)
|
(|$_{i \in R}$ |$_{j \in S}$!(ν $NB_{i,j}$) ($I_i,I_j,M_{i,j}$,I_i,I_j;$y1_{i,j}$).$\langle I_j,S,M_{i,j}$,$I_i,I_j,y1_{i,j}$,{$I_i,I_j,M_{i,j}$,$NB_{i,j}$ } KS_j [at $b1_{i,j}$ dest { $s2_{i,j}$ }] $\rangle$.(ν $MSG_{i,j}$) ($S,I_j,M_{i,j}$;$y2_{i,j}$,$y3_{i,j}$). decrypt $y3_{i,j}$ as {$NB_{i,j}$;$yk_{i,j}$ } KS_j [at $b2_{i,j}$ orig { $s4_{i,j}$ }] in $\langle I_j,I_i,M_{i,j}$,$y2_{i,j}$ $\rangle$.$\langle I_j,I_i,\{MSG_{i,j}$ } $yk_{i,j}$ [at $b3_{i,j}$ dest { $a3_{i,j}$ }] $\rangle$.0)
|
(|$_{i \in R}$ |$_{j \in S}$!(ν $K_{i,j}$) ($I_j,S,M_{i,j}$,I_i,I_j;$z1_{i,j}$,$z2_{i,j}$). decrypt $z1_{i,j}$ as {$I_i,I_j,M_{i,j}$;$zna_{i,j}$ } KS_i [at $s1_{i,j}$ orig { $a1_{i,j}$ }] in decrypt $z2_{i,j}$ as {$I_i,I_j,M_{i,j}$;$znb_{i,j}$ } KS_j [at $s2_{i,j}$ orig { $b1_{i,j}$ }] in $\langle S,I_j,M_{i,j}$,{$zna_{i,j}$,$K_{i,j}$ } KS_i [at $s3_{i,j}$ dest { $a2_{i,j}$ }] ,{$znb_{i,j}$,$K_{i,j}$ } KS_j [at $s4_{i,j}$ dest { $b2_{i,j}$ }] $\rangle$.0))

Table 5.13: LySa model of Otway-Rees Encryption with same key for initiating and responding

5.7.3 Insider Attacks

As well as situations where an attacker is trying to break in from the outside we must also consider insider attacks. In this scenario the attacker is an illegitimate principal, attempting to use his inside knowledge to gain more knowledge than they are allowed. A malicious insider is a powerful attacker in any situation. They do not need to break any perimeter defences and already have more knowledge than an outside attacker. A typical scenario may be an employee for a large company who is authorised to access details of his own clients but wants to be able to access all client information.

We also need to consider a scenario where an active principal is not the attacker but is leaking information to an outside attacker. This could be a deliberate, conscious move or perhaps they are doing so unwittingly through an attacker's machinations.

To model this unwanted flow of data we give illegitimate principals the index 0. All information with an index 0 is not restricted so our attacker model has access to it. If an attacker has this information they can then act as an illegitimate principal in order to attack a legitimate principal. We also modify the parallel composition |$_{i \in R}$ |$_{j \in S}$ to

allow for an initiator to begin a session with an illegitimate principal, for a responder to communicate to an illegitimate principal and for the server to communicate with illegitimate principals. The LySa model in Table 5.14 reflects this scenario.

let R $\subseteq \mathbb{N}$ in
let S $\subseteq \mathbb{N}$ in
((|$_{i \in R}$ |$_{j in S \cup \{0\}}$!(ν $NA_{i,j}$) ⟨I_i,I_j,$M_{i,j}$,I_i,I_j,{I_i,I_j,$M_{i,j}$,$NA_{i,j}$ } KS_i [at $a1_{i,j}$ dest { $s1_{i,j}$ }] ⟩.(I_j,I_i,$M_{i,j}$;$x1_{i,j}$). decrypt $x1_{i,j}$ as {$NA_{i,j}$;$xk_{i,j}$ } KS_i [at $a2_{i,j}$ orig { $s3_{i,j}$ }] in (I_j,I_i;$x2_{i,j}$). decrypt $x2_{i,j}$ as {;$xmsg_{i,j}$ } $xk_{i,j}$ [at $a3_{i,j}$ orig { $b3_{i,j}$ }] in 0)
|
(|$_{i in R \cup \{0\}}$ |$_{j \in S}$!(ν $NB_{i,j}$) (I_i,I_j,$M_{i,j}$,I_i,I_j;$y1_{i,j}$).⟨I_j,S,$M_{i,j}$,I_i,I_j,$y1_{i,j}$,{I_i,I_j,$M_{i,j}$,$NB_{i,j}$ } KS_j [at $b1_{i,j}$ dest { $s2_{i,j}$ }] ⟩.(ν $MSG_{i,j}$) (S,I_j,$M_{i,j}$;$y2_{i,j}$,$y3_{i,j}$). decrypt $y3_{i,j}$ as {$NB_{i,j}$;$yk_{i,j}$ } KS_j [at $b2_{i,j}$ orig { $s4_{i,j}$ }] in ⟨I_j,I_i,$M_{i,j}$,$y2_{i,j}$ ⟩.⟨I_j,I_i,{$MSG_{i,j}$ } $yk_{i,j}$ [at $b3_{i,j}$ dest { $a3_{i,j}$ }] ⟩.0)
|
(|$_{i in R \cup \{0\}}$ |$_{j in S \cup \{0\}}$!(ν $K_{i,j}$) (I_j,S,$M_{i,j}$,I_i,I_j;$z1_{i,j}$,$z2_{i,j}$). decrypt $z1_{i,j}$ as {I_i,I_j,$M_{i,j}$;$zna_{i,j}$ } KS_i [at $s1_{i,j}$ orig { $a1_{i,j}$ }] in decrypt $z2_{i,j}$ as {I_i,I_j,$M_{i,j}$;$znb_{i,j}$ } KS_j [at $s2_{i,j}$ orig { $b1_{i,j}$ }] in ⟨S,I_j,$M_{i,j}$,{$zna_{i,j}$,$K_{i,j}$ } KS_i [at $s3_{i,j}$ dest { $a2_{i,j}$ }] ,{$znb_{i,j}$,$K_{i,j}$ } KS_j [at $s4_{i,j}$ dest { $b2_{i,j}$ }] ⟩.0))

Table 5.14: Meta-LySa Model of Otway Rees including illegitimate principals

As must be expected at this point, all pretence of security is dropped with every message part deemed not confidential and every crypto-point assertion broken.

Violation of authentication properties (ψ)
(a1, CPDY), (a1, s2), (b1, CPDY), (b1, s1), (b3, a3), (b3, CPDY), (CPDY, a2), (CPDY, a3), (CPDY, b2), (CPDY, s1), (CPDY, s2), (s3, a2), (s3, b2), (s3, CPDY) (s4, a2), (s4, b2), (s4, CPDY),

Chapter 6

Automatic Generation of Protocol Implementation

6.1 Introduction

In this section we introduce the companion tool to Elyjah, named Hajyle (pronounced Hayley, and spelt like Elyjah backwards). It has been said that the biggest mistake which one can make when dealing with protocols is to attempt to write your own. If this is so, it makes sense to use an existing protocol which has already been analysed and approved. Being able to translate a validated LySa model of a protocol into a working implementation will help to reduce the number of errors made by developers attempting to implement a specification. Hajyle performs the inverse actions to Elyjah and automatically generates JaLAPI-based Java implementations of LySa code. While it is our belief that the translation performed by Elyjah is the more valuable by also having Hajyle the two tools form a symbiotic relationship as seen in Figure 6.1.

A big flaw in only having a Hajyle-like specification-to-implementation tool is that a developer could not make any changes to the generated program code and be confident that it is still a valid implementation of the specification. By having a tool like Elyjah developers can continue to make changes and receive assurances as to the security properties of the implementation.

Additionally the tools can together be used to validate the consistency of the translations. Namely, by starting with a LySa model and then using Hajyle followed by Elyjah on the generated Java file the resulting LySa file should be equivalent to the original. The same will not necessarily be true if starting and finishing with Java as certain details such as variable names will be lost when converting between Java and

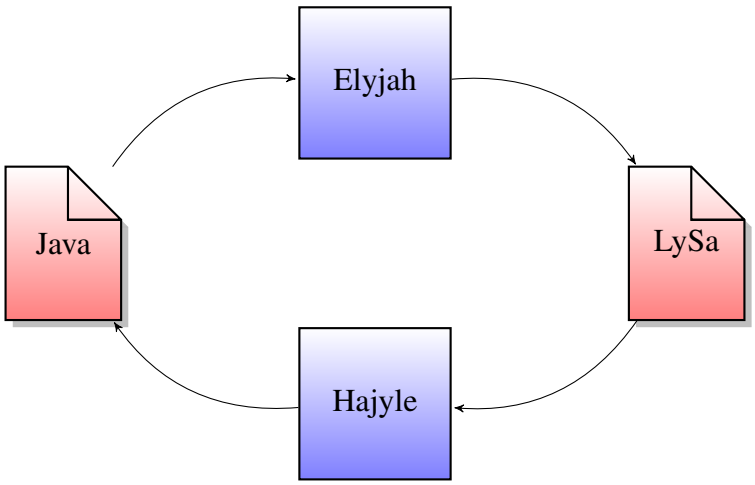


Figure 6.1: Elyjah - Hajyle relationship

LySa so these names have to be automatically generated as part of Hajyle's process. Despite these minor differences the two Java programs will implement the same protocol.

6.2 Requirements of the Hajyle Software

The chief requirement of Hajyle is that it generates accurate Java implementations of the supplied LySa protocols. In order that we can use Elyjah to validate this translation, and indeed any changes made to the generated file, Hajyle should produce Elyjah-compatible implementations which use JaLAPI wherever possible. This provides the added benefit of simplifying the produced code so that it is easier to understand and a developer can more easily make any amendments without disturbing the protocol-related parts if they do not desire to do so.

Connected to this is the requirement that the produced Java programs should be fully runnable without any further modification from the developer. Developers should be able to make careful changes to the code and additionally to make changes to the JaLAPI implementation. Obviously this may break the program but it is left to the developer to make these mistakes on their own.

Speed is not so much a priority for Hajyle as it is for Elyjah as the normal use of this tool would be for it to be used once in a development cycle to create a skeleton prototype code that would then be amended and checked using Elyjah. Despite this, Elyjah has shown that very quick translation in the opposite direction is possible so we

aim for similar performance from Hajyle.

Hajyle is designed to work with standard LySa models that explicitly state protocol and key names and do not use bi-directional principal names such as those seen introduced in Section 5.7.2. Meta-LySa models which only add definitions of the principal numbers to the deployment scenario as in Section 5.7.1 are processed by dropping the Meta-LySa annotations and generating Java based on the plain LySa model that is left.

While it is possible to add Meta-LySa annotation to LySa models by converting a LySa model to Java using Hajyle and then using Elyjah to generate a Meta-LySa model, it is a rather inelegant solution and could best be described as an un-documented feature of these tools. It is not the aim of this work to find or rediscover existing flaws in protocols.

6.3 Hajyle Implementation

6.3.1 Preliminary Tasks

Hajyle uses the same LySa parser created for the LySa Toolkit in Eclipse presented in Section 3.1.

The first task Hajyle undertakes is to identify all the keys used in the protocol. We use the same technique described in Section 4.2 to accomplish this. This does mean that any protocol which establishes and shares a key but never uses it for encryption is not perceived to be a key by Hajyle but instead a plain message part. However as Section 5.6.2 notes, it is worth appending such a message exchange in the LySa model so that the LySatool can analyse the confidentiality of this communication and whether there are any violations of crypto-points.

The next task is to match up all send processes with the corresponding receive process. This allows Hajyle to create a mapping between what the different principals call the same objects. For example in the following LySa model principal *A* and principal *B* refer to the same message part as *msg* and *x* respectively.

$$(\nu\ msg)\ \langle A,B,msg\rangle.0$$
$$|$$
$$(A,B;x).0$$

6.3.2 Generating Set Up Class

The final task before analysing each principal is to identify the roots of each principal and determine the name of the principal. This is done by looking through the model for a send or receive process and taking the first part of the tuple, the sender. We also examine any keys shared with all principals before the protocol begins. We use all this information to construct the set up class. We start by initialising the `Principal` classes used to model the LySa principals. Firstly the name of the Java file we are going to be using is determined by taking the file-name of the LySa file, removing the ".lysa" suffix and changing the rest of the name to lowercase except for the first letter. Our class file-names are then constructed from this name appended with the principal name. We then append the code to initiate and register these classes with a `Network`. Finally we have to detect any new keys established before the protocol begins. The appropriate generate method code is appended to create a key variable and use the `shareKey` method in the `Network` class to register the key with all principals. Finally we start the threads using the class's `start` method, this has to occur after any keys have been shared with all principals.

6.3.3 Generating Principal classes

Having already identified the principal roots in order to determine principal names we analyse each principal in turn starting with the root node. The first task is to construct the method header and standard code used to initiate a global variable for the `Network` object and create a constructor method which fills this variable.

It must then be determined if there is a requirement for a `run` method in this class. To do this, the LySa process is examined to find out whether a send or a receive process comes first. If a send process comes first, then this class requires a `run` method which sends this first message and creates any required keys or other message parts.

Following this we generate the `processIncoming` method and close the class.

6.3.3.1 Creating the `run` Method

There are two process types that need to be encapsulated in a `run` method, processes relating to the ν operator and any send processes that come before a receive process. The ν processes are worked out first by finding all LySa processes involving the new keyword. These may either be keys or message parts such as a nonce or secret information. As part of our preliminary task we already identified the names of keys and

also a matching of received variables to sent information. This allows us to determine whether the variable being created in the LySa process is used as a key in the current principal or another one after it has been sent. If the variable is a key then we create either a symmetrical key or asymmetrical keypair and register this key with the key store. If the variable is not a key then we create a new variable and use the JaLAPI method `generateMessage` so that if the generated code is later supplied to Elyjah, the output file will feature the LySa ν process. However, this may not be an accurate notion of the protocol's intent, for example the variable may actually be a random nonce or indeed a key that is not used to encrypt or decrypt anything in the protocol. These are situations where it is impossible or at least foolish to try to divine any further information from the LySa model.

After initiating any variables required, the Java code representing a LySa send process can be created. For each LySa node SendProcess, Hajyle creates a new `Message` object with a unique variable name. Each child of the SendProcess node represents a message part to be added to the new `Message` object. The first two nodes represent the sender and receiver in LySa so we ignore these and for each of the remaining nodes, we determine whether we are dealing with a key, a message part or an encrypted message block. If we are adding a key then we have to insert Java code to retrieve the correct `Key` from the keystore, storing it in a variable which we then turn into a `String` via JaLAPI's `sendKey` method. If the message part we are adding is neither a key nor an encrypted block then we must determine if this is a variable which has some value previously assigned to it or a constant such as a name. We do this by determining if the message part has previously appeared in a ν process or as part of the variable instantiation phase of a receive process. If so then this message part is a previously assigned variable and should be appended in the Java code as such or is a constant which will be added in quotes. The final option is that we are adding an encrypted portion. This is assembled much like a send process, beginning with creating a new `Message` object for the encrypted portion. Then each message part is processed in turn as either a key, message part or encrypted portion. Once the `Message` object has been populated, the additional step of encrypting it is performed. Firstly the correct key is retrieved from the keystore then we invoke the parent `Message` object's `add` method with the parameter of the returned text of an `encrypt` method call with our new `Message` and `Key` object. If crypto-points were present on the encryption these are added as parameters to the `encrypt` method call. This process is summarised in Figure 6.2.

Once the `Message` object is fully constructed we append the Java code to send the

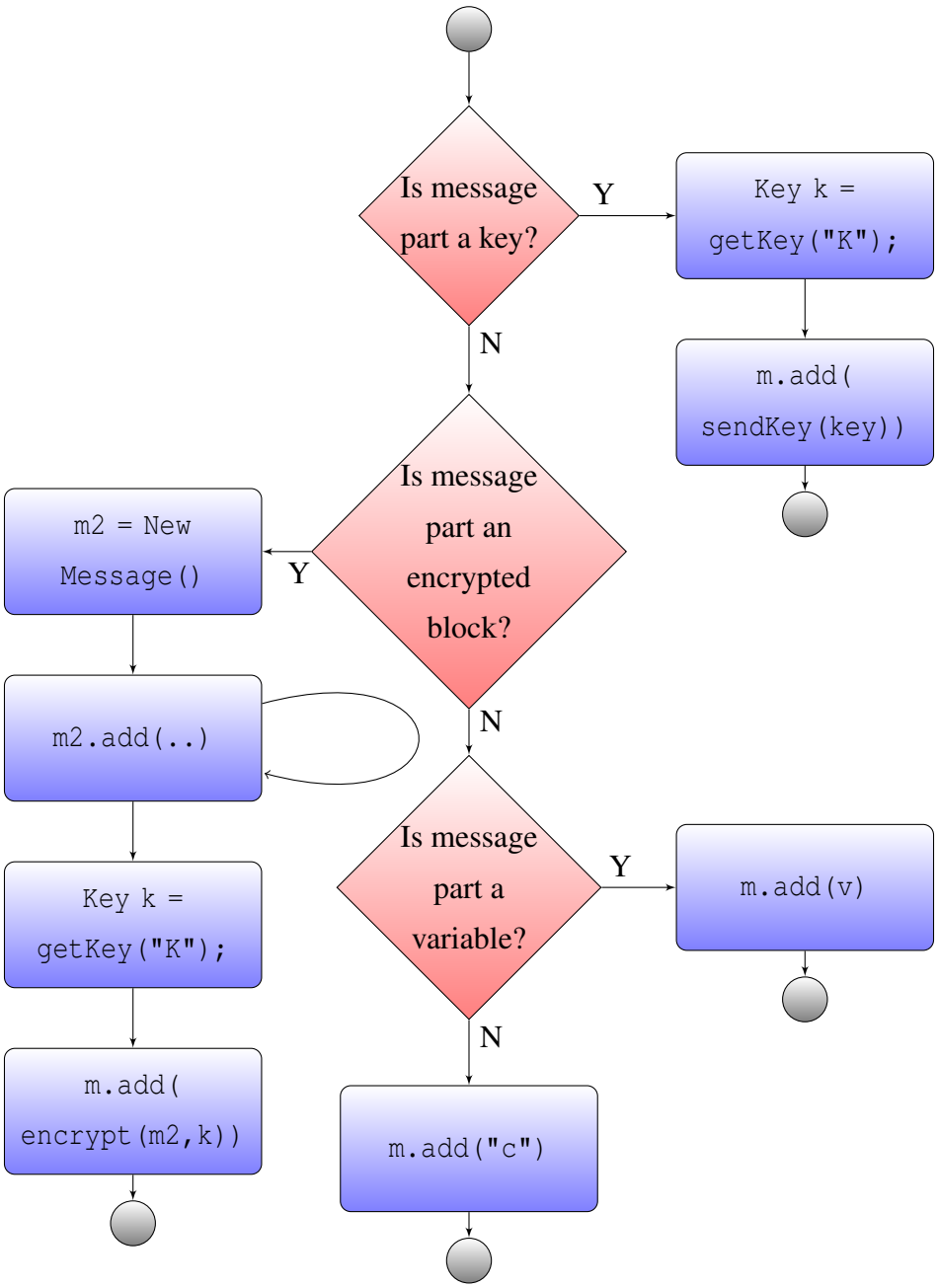


Figure 6.2: Add Message Part Flowchart

message to the correct principal, namely the second tuple in the LySa model.

6.3.3.2 Creating the `processIncoming` Method

Like the `run` method, dealing with the `processIncoming` method starts with constructing the correct Java method header and starting the `switch` statement. We then search for each receive process and for each one we find we find all new, decrypt or

send processes between the previous receive process and the receive/principal. We create a `switch` block for each of these process blocks. The LySa process for receiving a message is split into two parts, one for constants which are checked against expected values and one for variable instantiation. Starting with the receive process we construct the Java for checking message parts. The first two tuples have been reserved for the sender and the receiver so we append the JaLAPI `getSource` and `getDest` methods and the JaLAPI `getNext` method for dealing with the remaining constants. For the variable instantiation, we have to additionally check whether the incoming message part is a key. If so instead of storing the message part in a string, we determine the type of key and use the correct JaLAPI receive `Key` method to convert the `String` representation into the correct format. This `Key` is then registered with the key store using the name of the message part in the LySa process.

Unlike encryption which is handled within a send process, decryption in LySa is a disjoint process which occurs after the encrypted string has been stored in a variable. Firstly the required `Key` is retrieved from the keystore and a uniquely named `Message` object is created and populated with the JaLAPI `decrypt` method which takes in an encrypted message block as a `String` and a `Key` and returns a populated `Message` object. If the LySa decrypt process had crypto-point annotations then these are captured in the Java as additional parameters to the `decrypt` method. If the annotation for the origin of the encrypted message is a list then an `ArrayList` object is created and populated accordingly. Much like encryption is similar to assembling a message, the process of decryption is very similar to processing an incoming message. The only real difference is that there is no checking of source or destination for the `Message`. Aside from this constant checking and variable instantiation is handled exactly the same as an incoming message.

At this point, the techniques employed in Section 6.3.3.1 are employed again to generate the Java for the ν, and send processes.

6.4 Worked Examples

6.4.1 Simple Symmetric Encryption

In order to show an overview of this tool we will present a simple example where two principals already share a symmetric key and use this to send a message securely between them. We have to generate some overhead Java code for the setup of the

(ν K) (
(ν msg)⟨A,B,{msg}$_K$ [at a dest {b}]⟩.0
|
(A,B;x). decrypt x as {;y}$_K$ [at b orig {a}] in 0
)

Table 6.1: LySa Model of Simple Symmetric Encryption protocols

protocols. Hajyle generates the Java code for initialising two principals like the code written in Section 5.6.1 so all that is left is to generate the Java code that implements the LySa (ν K) process. Looking through the rest of the protocol it is clear that *K* is a key as it is used to both encrypt and later decrypt part of a message. Thus we use the JaLAPI method to generate a symmetric key and then store it in both principal's key store. To do this succinctly Hajyle uses the JaLAPI `network`'s object `shareKey` method.

```
SecretKey K = generateSharedKey();
net.shareKey(K, "K");
a.start();
b.start();
```

The next line of Java code is the start of the protocol. For this reason it is placed in principal *A*'s `run` method. Firstly it is determined that the LySa (ν msg) process does not represent the creation of a new key so we use the JaLAPI `generateMessage` method. In this scenario, this is the correct move, however if *msg* was to be used as a nonce, Hajyle would not be able to realise this. Hajyle then initialises two `Message` objects one for the variable `msg` containing the protocol's secret communication and one for storing the encrypted representation of this first `Message`.

```
public void run() {
        msg = generateMessage("MESSAGE");
        Message message = new Message();
        Message encryptedPart = new Message();
        encryptedPart.add(msg);
        SecretKey key_2 = (SecretKey) getKey("K");
        message.add(encrypt(encryptedPart, key_2, "a", "b"));
        // Send message
        net.send("B", message);
```

```
}
```

The final line in the LySa model is implemented in the `processIncoming` method of principal *B*. Due to the structured nature of the JaLAPI code, Hajyle can insert automatically generated comments.

```
public void processIncoming(Message msg) {
        switch (receivedNum) {
        case 0:
                receivedNum++;
                // Check claimed source
                check(msg.getSource(), "A");
                // Check intended destination
                check(msg.getDest(), "B");
                // Check message contents
                // Add variable instantiation
                x = msg.getNext();
                // Decrypt message parts (1)
                SecretKey key_2 = (SecretKey) getKey("K");
                Message decrypted = decrypt(x, key_2, "b", "a");
                y = decrypted.getNext();
                break;
        }
}
```

6.4.2 Otway-Rees Protocol

In order to examine any differences between hand-written protocols and the implementations that Hajyle generates we re-examine the Otway-Rees protocol last seen in Section 5.6.2. Indeed, we will be using the LySa generated by Elyjah which was presented in Table 5.10 so that the output of Hajyle may be as close to the original input to Elyjah as possible.

The first difference is in the set up class, while in our original code we used the principal's `shareKey` method to register the symmetric keys with the principals Hajyle uses the `shareKey` method from the `Network` object. This makes for simpler Java code, sharing a key only requires this one JaLAPI method invocation to register the key with principal's key store. This does result in more principals having access to the key than the developer may wish but is, in truth, a more accurate representation of the

LySa model.

Handwritten Code	Hajyle-Generated Code

`SecretKey keyA = generateSharedKey();`
`SecretKey keyB = generateSharedKey();`
 	
`a.shareKey(keyA, "KAS");`
`s.shareKey(keyA, "KAS");`
`b.shareKey(keyB, "KBS");`
`s.shareKey(keyB, "KBS");`
	`SecretKey KBS = generateSharedKey();`
	`net.shareKey(KBS, "KBS");`
	`SecretKey KAS = generateSharedKey();`
	`net.shareKey(KAS, "KAS");`

Another difference between the two is that Hajyle treats the session counter *M* as a constant as the protocol makes it clear all participants know this message part before the protocol starts. Hayjle also uses the JaLAPI `generateMessage` method instead of the correct `generateNonce` method for the nonces *NA* and *NB*. As explained before, this is because LySa uses the same process for both.

Handwritten Code	Hajyle-Generated Code

`NA = generateNonce();`
`Message v = new Message();`
`v.add(M);`
`v.add("A");`
`v.add("B");`
 	
`Message vEncoded = new Message();`
 	
`SecretKey keyAS = (SecretKey) getKey("KAS");`
`vEncoded.add("A");`
`vEncoded.add("B");`
`vEncoded.add(M);`
`vEncoded.add(NA);`
 	
`v.add(encrypt(vEncoded, keyAS, "a1", "s1"));`
`net.send("B", v);`
	`NA = generateMessage("MESSAGE");`
	`Message message = new Message();`
	`message.add("M");`
	`message.add("A");`
	`message.add("B");`
	`Message encryptedPart = new Message();`
	`encryptedPart.add("A");`
	`encryptedPart.add("B");`
	`encryptedPart.add("M");`
	`encryptedPart.add(NA);`
	`SecretKey key_2 = (SecretKey) getKey("KAS");`
	`message.add(encrypt(encryptedPart, key_2, "a1", "s1"));`
	`// Send message`
	`net.send("B", message);`

Aside from these differences which are consistent throughout the code, there are no further differences between the handwritten code and the code generated by Hajyle. Indeed both have the same Java trace when run. Additionally, by running Elyjah on both of these files returns the same LySa model introduced in Table 5.10. This supports our hypothesis that converting a protocol model into Java and back into LySa returns the original model.

6.5 Performance

Although the performance aspect of Hajyle is less crucial than it is for Elyjah, it is still reassuring to know that the process is quick enough that that it will not delay development in any way. The table below shows the times for Hajyle to generate Java implementations for the same protocols as seen in Section 5.5.1. Differences in line counts to these handwritten programs can be attributed to various comments, line

breaks or syntactical differences such as whether a `Message` object was encrypted and added to the parent `Message` in two separate lines of code or the `encrypt` method invocation being called as the parameter for the `add` method.

Protocol	Generated Implementation			Timings
	LoC	Messages	Principals	Hajyle
Needham-Schroder	227	6	3	0.046s
Otway-Rees	222	5	3	0.044s
Wide Mouth Frog	144	3	3	0.031s
Yahalom	184	4	3	0.033s

Chapter 7

Elucidation of Formalisation

In Chapter 5 and Chapter 6 we presented a pair of tools that allow a developer to translate between a Java implementation and a LySa representation of a cryptographic protocol. In this Chapter we examine the relationship between our chosen languages.

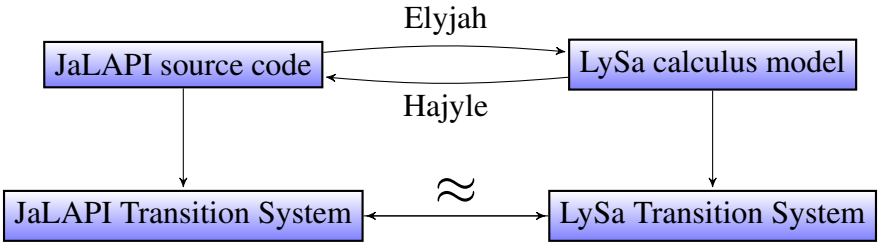
The fundamental problem faced when attempting to discuss the equivalences between Java programs and LySa models is that while LySa is a context specific language designed for the sole task of modelling cryptographic protocols, Java is a general purpose programming language. This means that each LySa process takes multiple lines of code when implemented in Java. The act of implementation itself adds further lines of code to a Java program. LySa models do not specify implementation details such as the encryption algorithm or the communication method while these details are required in the Java. Luckily much of this information can be hidden inside the implementation of JaLAPI and does not take up space in the protocol implementation file. This additionally fits with the concept of black-box encryption as espoused by process calculi like LySa.

We start by deconstructing LySa processes in order to break them down into the individual steps that JaLAPI can accomplish. As a starting point we use the LySa reduction rules presented in Figure 7.1 as originally defined in [21].

Section 7.2 then gives a formalisation of JaLAPI method invocations using operational semantic rules. Together with the correspondence diagrams in Section 7.1 we can then construct operational semantic rules for Java implementations of LySa models which mirror those in Figure 7.1.

The diagrams in Section 7.1 show how combining several rules allow us to equate LySa processes with JaLAPI. Transitions between LySa terms or multiple-part JaLAPI transitions are on the right and these are broken up into sequences of abstract transitions

which are shown on the left. As the LySa terms are much more concise than Java programs, LySa processes are broken up into several sub-diagrams. Also in these situations, the more general rule is on the right and the detailed break-down on the left. This will demonstrate that the transition systems of these two formalisations are comparable and models described in both are equivalent, in the sense shown below.



The desired relationship between the JaLAPI and LySa Transition Systems represented in the diagrams in this chapter by the $\approx$ symbol can be defined as preserving these rules:

- If we have Java program J, whose translation $Elyjah(J)$ has security properties P as reported by $LySatool(Elyjah(J))$ then execution of J will also have properties P.
- If we have Java program J, whose translation $Elyjah(J)$ has protocol narration N as reported by $AnaLySa(Elyjah(J))$ then execution of J will exhibit protocol narration N.
- If we have LySa model L, which has security properties P as reported by $LySatool(L)$ then translation $Hajyle(L)$ will also have properties P.
- If we have LySa model L, which has protocol narration N as reported by $AnaLySa(L)$ then translation $Hajyle(L)$ will exhibit protocol narration N.

7.1 LySa - JaLAPI Relations

7.1.1 ANEW and NEW Rules

The simplest LySa rules to express with JaLAPI are the LySa NEW and ANEW rules for generating new message parts or keys. The NEW rule is used to create new keys, nonces and message parts while the ANEW rule is exclusively used for generating asymmetric key pairs. To achieve this process with JaLAPI is a multi-step procedure.

COM

$$\langle V_1,...,V_k\rangle.P_1|(V_1,...,V_j;x_{j+1},...,x_k).P_2 \;\rightarrow\; P_1|P_2[x_{j+1} \overset{\alpha}{\mapsto} V_{j+1},...,x_k \overset{\alpha}{\mapsto} V_k]$$

SDEC

$$decrypt\ \{V_1,...,V_k\}_{V_0}\ as\ \{V_1,...,V_j;x_{j+1},...,x_k\}_{V_0}\ in\ P \;\rightarrow P[x_{j+1} \overset{\alpha}{\mapsto} V_{j+1},...,x_k \overset{\alpha}{\mapsto} V_k]$$

ADEC

$$decrypt\ \{|V_1,...,V_k|\}_{m^+}\ as\ \{|V_1,...,V_j;x_{j+1},...,x_k|\}_{m^-}\ in\ P \;\rightarrow P[x_{j+1} \overset{\alpha}{\mapsto} V_{j+1},...,x_k \overset{\alpha}{\mapsto} V_k]$$

ASIG

$$decrypt\ \{|V_1,...,V_k|\}_{m^-}\ as\ \{|V_1,...,V_j;x_{j+1},...,x_k|\}_{m^+}\ in\ P \;\rightarrow P[x_{j+1} \overset{\alpha}{\mapsto} V_{j+1},...,x_k \overset{\alpha}{\mapsto} V_k]$$

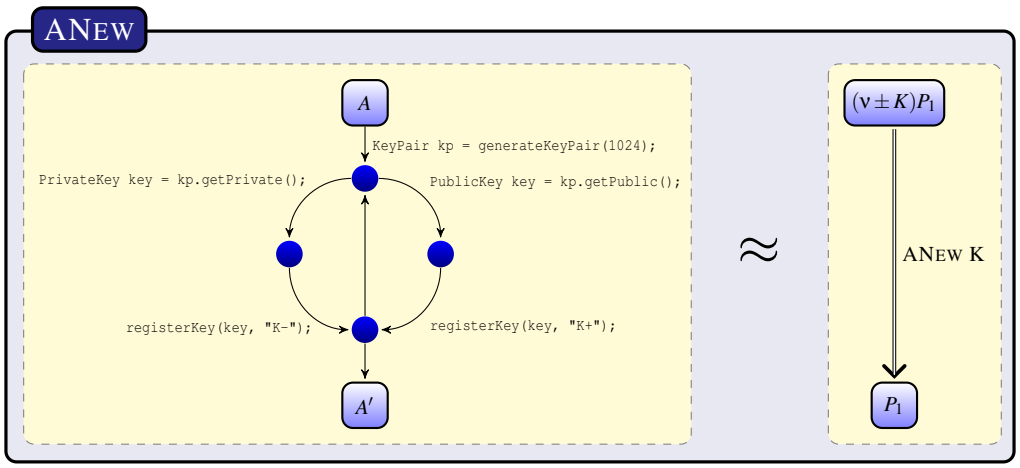
$$\text{NEW}\ \frac{P \rightarrow P'}{(\nu\ n)P \rightarrow (\nu\ n)P'} \qquad \text{ANEW}\ \frac{P \rightarrow P'}{(\nu \pm\ m)P \rightarrow (\nu \pm\ m)P'}$$

$$\text{PAR}\ \frac{P_1 \rightarrow P_1'}{P_1|P_2 \rightarrow P_1'|P_2} \qquad \text{CONGR}\ \frac{P \equiv P''\ \ P'' \rightarrow P'''\ \ P''' \equiv P'}{P \rightarrow P'}$$

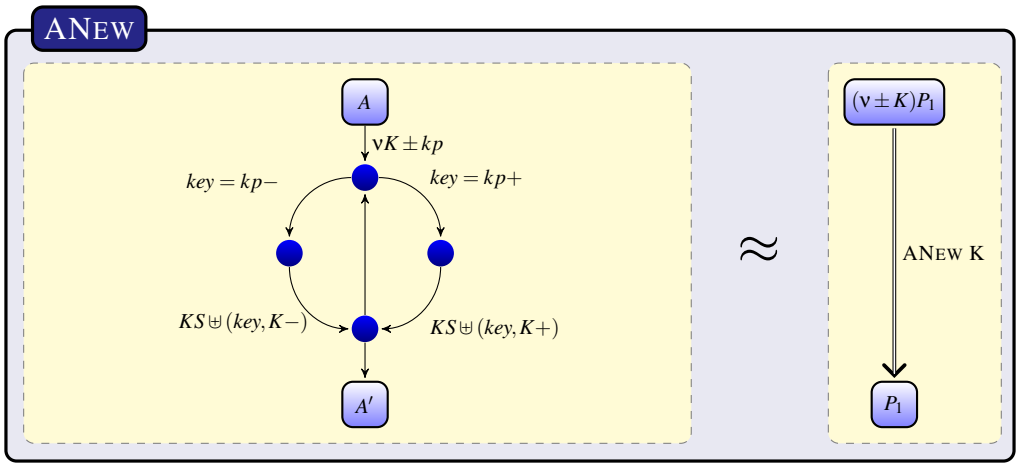
Figure 7.1: The reduction relation P→ P'.

JaLAPI has a method to generate a Java `KeyPair` object which itself has methods for retrieving corresponding private and public keys. Additionally, as mentioned when discussing Elyjah and Hajyle, we require keys to be registered with a keystore such that there is a fixed name which is used to refer to keys in a LySa model. Thus in order to replicate the LySa ANEW rule there must be several method calls; generating the pair, taking either the public or private key and then registering the key with the keystore. In this instance it is not necessary for both paths to be followed as it is entirely possible that only one of the keys is needed in a Java implementation although it is possible that both key parts will be used. This is shown in the following diagram.

In this and subsequent diagrams we use a normal arrow when the process cannot be broken down into several Java instructions. Equally we use a double arrow to denote that the arrow signifies several combined processes that will be explained later. A dotted arrow indicates “and so forth” in the same manner that an ellipsis does in mathematical notation.

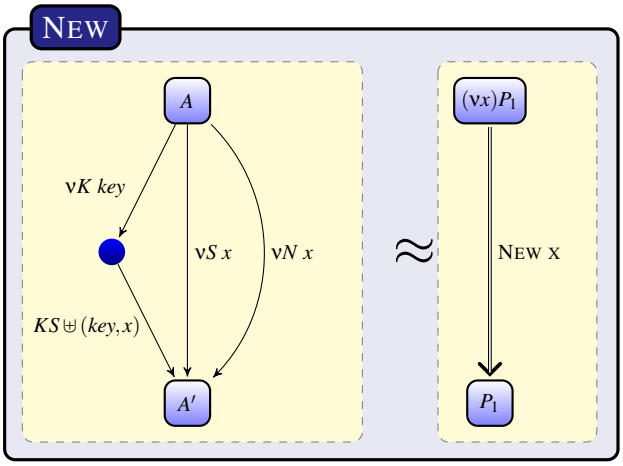


In an effort to simplify these diagrams we introduce a labelling system for transitions. These labels borrow syntactically from both LySa and Java where appropriate. Conciseness is the key aim in generating these labels so when LySa provides a syntactic framework applicable for the purpose of a rule this was used. For this reason, we use a variation of the LySa ν process for new objects. We use the label $\nu K \pm kp$ instead of the full `generateKeyPair` method call. For retrieving a key from a key pair we use Java-like variable-assignment syntax coupled with representing public and private keys with + and - characters, for example, $key = kp-$ assigns $kp-$ to key. For adding a new tuple to the keystore we use the $\uplus$ character, for example $KS \uplus (key, k)$ registers key K in the keystore under the name key. These simple labels show both the commands as well as the variables used. This gives us a simplified and updated figure as below.



The difficulty when attempting to represent the more general NEW rule with JaLAPI methods is that there are several possible meanings to the LySa process $(\nu\, x)$. It is not clear whether the generated x is a key, a nonce or simply a plain string. There are

JaLAPI method calls for generating each of these message parts which shall be represented by the labels νK, νN and νS respectively. Assuming the item is a key then as above, it must be registered with the keystore.

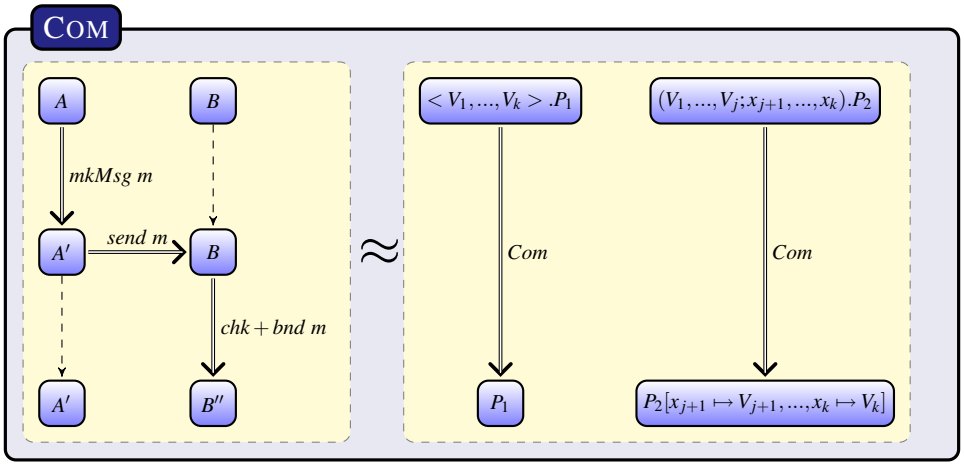


7.1.2 COM Rule

We continue by trying to capture the rules required to express the LySa notion of communication as given in the LySa reduction rule COM in Figure 7.1.

$$\langle V_1,...,V_k\rangle.P_1 | (V_1,...,V_j;x_{j+1},...,x_k).P_2 \quad \rightarrow \quad P_1 | P_2[x_{j+1} \overset{\alpha}{\mapsto} V_{j+1},...,x_k \overset{\alpha}{\mapsto} V_k]$$

The LySa process that this rule represents embodies message creation, composition and communication in one process. This process is split over multiple method invocations in Java code. The COM rule can be thought of in three distinct phases, message composition, communication and processing of the incoming message. The distinction is slightly unclear as composing and sending a message as well as receiving and processing a message both occur in one step.



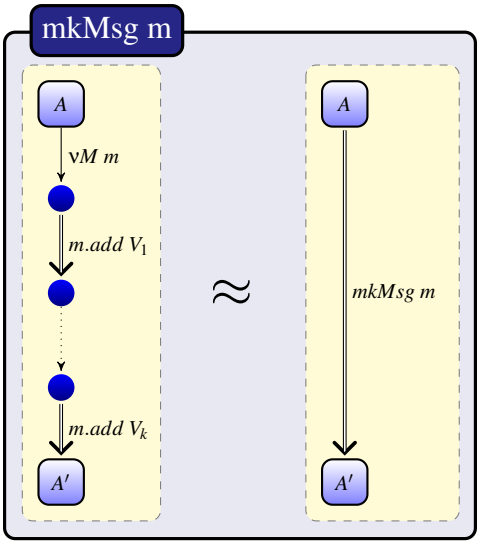
Equivalent pseudo-Java-code for these two processes is shown below.

run()	**processIncoming()**

`Message m = new Message{};`
`m.add(`V_1`);`
`...`
`m.add(`V_k`);`
	`check(msg.getSource(), `V_1`);`
	`check(msg.getDest(), `V_2`);`
	`check(msg.getNext(), `V_3`);`
	`...`
	`check(msg.getNext(), `V_j`);`
	`String `x_{j+1}` = msg.getNext();`
	`...`
	`String `x_k` = msg.getNext();`

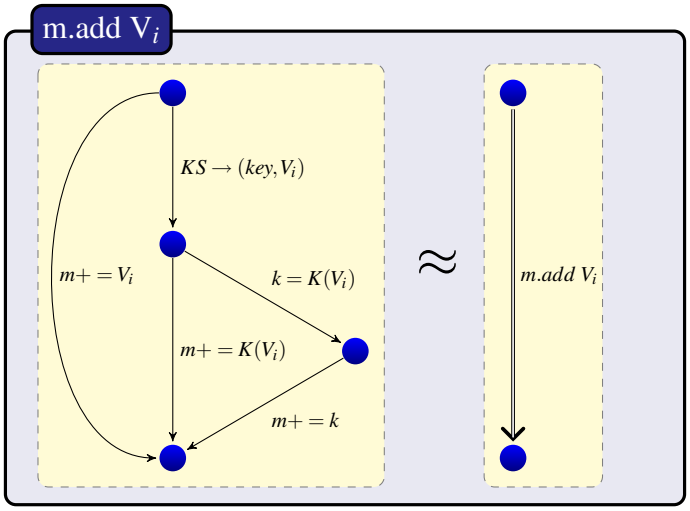
7.1.2.1 Message Creation and Composition

Message creation in JaLAPI means creating a new `Message` object and adding message parts individually. For an n element message, it thus takes at minimum $n + 1$ method calls to express using JaLAPI. Following the convention introduced in Section 7.1.1 we use a variation of the LySa ν process to denote the generation of a new `Message` object, in this instance M is appended to the ν character. We use Java syntax for representing adding elements to the `Message`. As this process has many possible implementations we use a double arrow to signify it can be broken into child processes. We will present a fuller and more complete explanation on this after the following diagram



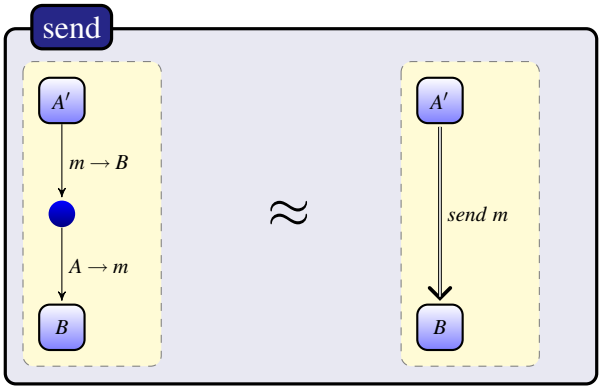
There are additional processes used, for example, in the case where the message part is a `Key`. This means that adding a message part to a `Message` object is a more complicated procedure than it might at first appear. The potential options available are either we are adding a `String`, be it a literal, variable or an encrypted block; or we want to add a `Key`. This key must be first retrieved from the keystore and converted into a transmissible format for communication. In Section 7.1.1 we presented the

label for adding items a keystore: $KS \uplus (key, K)$. The label for retrieval has a similar format with the $\uplus$ label replaced with an arrow denoting the key and label coming from the keystore. There are two options for then adding a `Key` to a `Message`. Although semantically identical they differ in syntax. Either the key is converted to a `String` in one line and this `String` added to a `Message` or by using the `sendKey` method directly as the parameter for the `add` method. When adding elements to a `Message` we use the following concise Java-like syntax to describe adding an element x to a `Message` m: $m+=x$. Finally we have the notation $K(x)$ to describe a `Key` which is in a transmissible format. For example $m+=K(key)$ means adding a `String` representation of a `Key` *key* to a `Message` m.



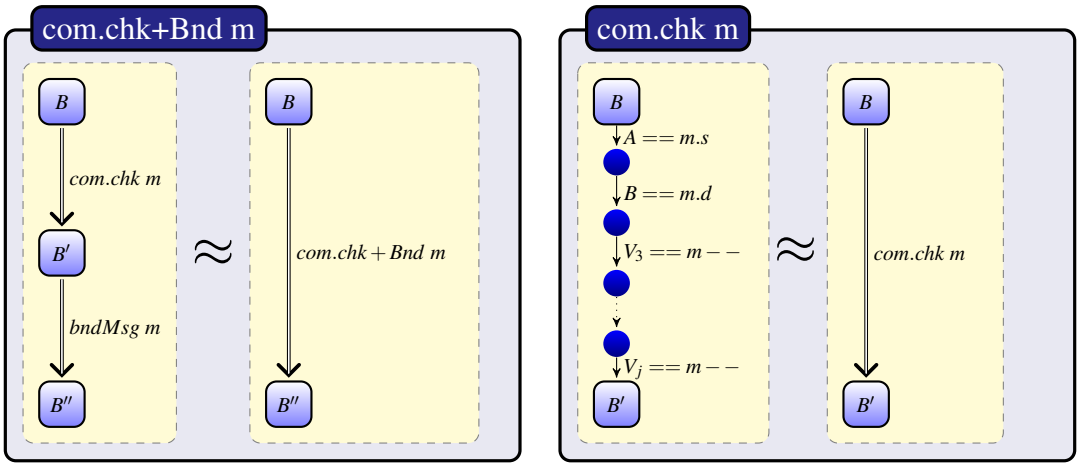
7.1.2.2 Sending and Receiving Messages

In the COM diagram in Section 7.1.2 we label one of the transitions as send. This is a two-part process, in our source principal the JaLAPI `send` method is called and in our destination principal we have the method header for the `processIncoming` method. The label for our send transition denotes a message, m, being sent to a principal B. The receive transition label takes the form of source and incoming `Message`.

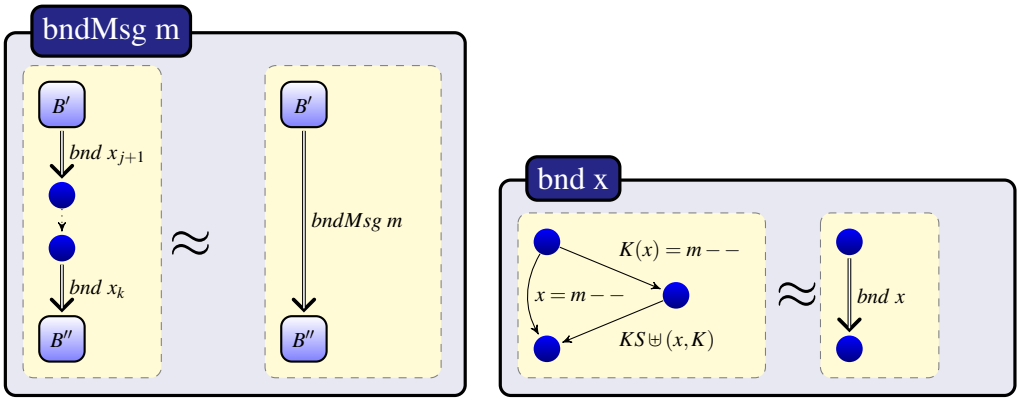


7.1.2.3 Process Received Message

In LySa, the receipt of a message happens at the same time as message processing. This processing is achieved by pattern-matching on one section of the message and variable instantiation on the latter. This process is illustrated by the com.chk+Bnd m diagram. Additionally in LySa, it is typical for the first two elements of a message to be the source and destination. In our implementation these are automatically appended to messages in the send process. There exist JaLAPI methods for checking the source, destination and subsequent message parts. Checking the content of the message is done one tuple at a time. The transition labels for these JaLAPI method calls take the form of comparison statements for Java primitive types. We use the notation $m--$ to represent removing a tuple from a `Message` m in a FIFO sequence to allow processing of the later parts.

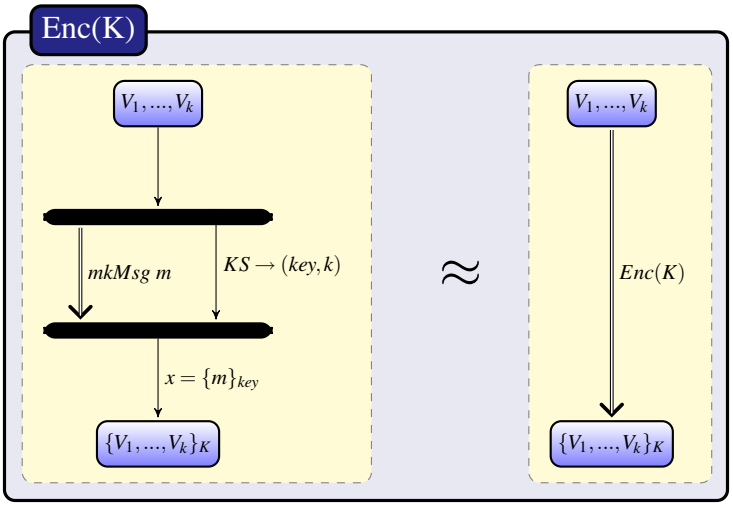


As detailed in the com.Chk+Bnd m diagram, after checking the first half of the message variable binding then takes place in a similar manner to the previous message checking. Each subsequent message part is checked in turn as seen in bndMsg m. The incoming message part could potentially be a standard message part such as a principal name or nonce, an encrypted portion or a string representation of a key. In this instance the key must also be registered with the keystore.



7.1.3 Encryption

Encrypted message parts are a base LySa term that are used in message composition. In order to implement this in Java with JaLAPI, we have to construct a new `Message` object and add elements to this `Message`. Additionally the required key has to be retrieved from the keystore. Only when both of these steps have been accomplished can the encryption itself be performed. The diagram below makes it clear that the two required steps can be performed in any order, although in fact the more general scenario is applicable where the key retrieval can be performed at any point during the ongoing process of preparing the message block. We use the LySa encryption notation for the transition label of the JaLAPI `encrypt` method, meanwhile we have already shown the breakdown for how a `Message` is generated and how a key is retrieved from the keystore. The generated `String` can then be added to a `Message` so it can be sent to another principal. The process for asymmetrical encryption or generating a signature is identical, with the only difference being the use of the appropriate key.



7.1.4 Decryption

The LySa rules featured in Figure 7.1 include the variations of a standard decryption process. These rules feature built in encryption in a manner not used in actual LySa models. The SDEC rule, for example, would typically be found in a scenario such as:

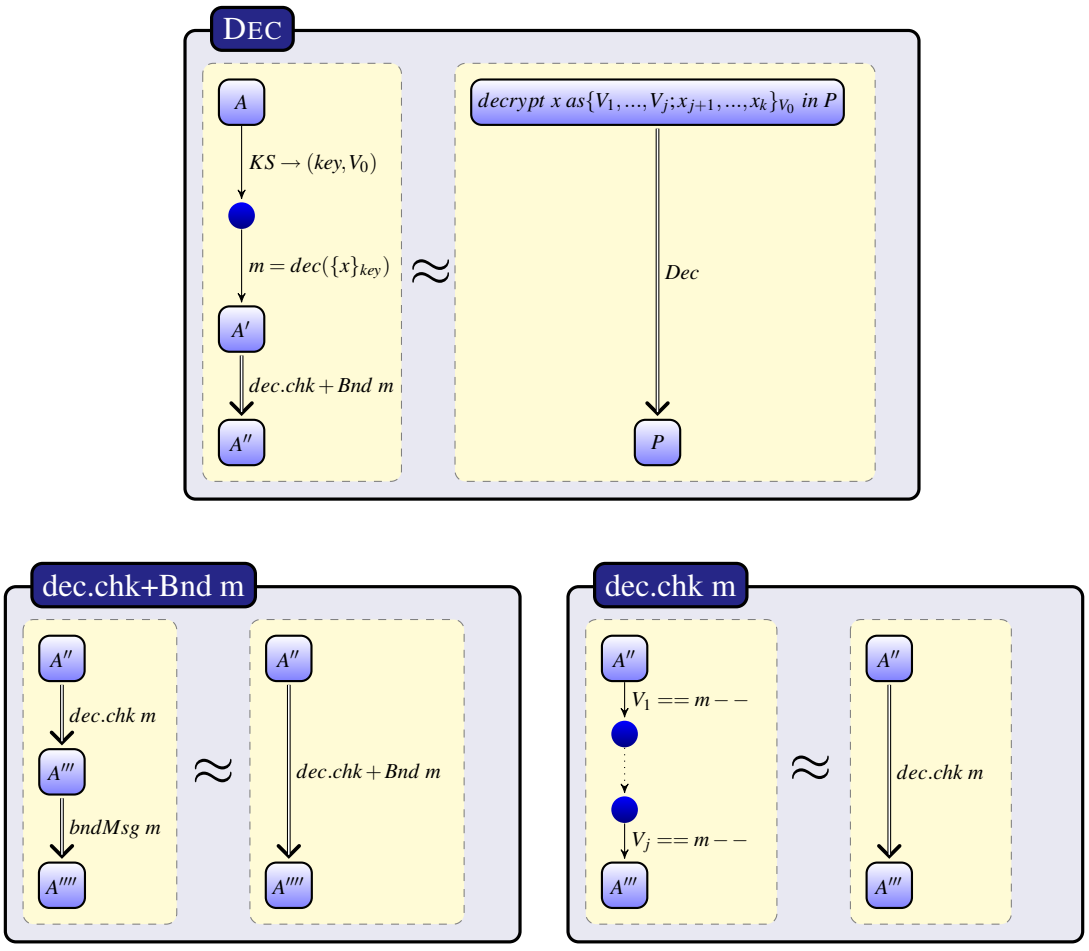
$$\langle\{V_1,...,V_k\}_{V_0}\rangle.P_1 \mid (;x_0)\ .\ decrypt\ x_0\ as\ \{V_1,...,V_j;x_{j+1},...,x_k\}_{V_0}\ in\ P_2$$
$$\rightarrow P_1 \mid P_2[x_{j+1} \overset{\alpha}{\mapsto} V_{j+1},...,x_k \overset{\alpha}{\mapsto} V_k]$$

However, it is clear that the rules as presented in Figure 7.1 are much more concise. Rather than repeat ourselves we shall focus on the decryption portion, for example

SDEC

$$decrypt\ x_0\ as\ \{V_1,...,V_j;x_{j+1},...,x_k\}_{V_0}\ in\ P\ \rightarrow P[x_{j+1} \overset{\alpha}{\mapsto} V_{j+1},...,x_k \overset{\alpha}{\mapsto} V_k]$$

where we take as given that x_0 has the form $\{V_1,...,V_K\}_{V_0}$. The diagram below shows the deconstructed JaLAPI decryption process. The first stage is retrieval of a key from the keystore. The decryption itself takes place next which converts a `String` into a `Message` object. The label for this transition borrows from Java as with the built-in pattern matching, the LySa syntax for decryption is longer. The JaLAPI pattern matching then takes place over the generated `Message`. The check portion is similar to that of incoming messages but does not check source and destination first. Variable instantiation is identical to the breakdown previously defined in the bndMsg m figure in Section 7.1.2.3.

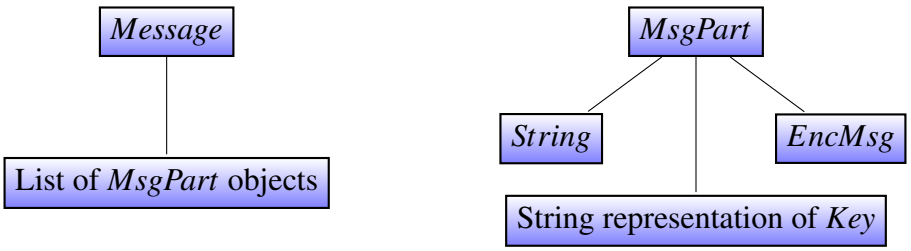


The breakdown for both ADEC and ASIG are identical, only using the correct key to represent the relevant rule.

7.2 Formalisation of JaLAPI

We want to show that the same protocol can be described as both a LySa model and a Java program using JaLAPI. Thus when trying to define the formalisation that allows us to reason about our equivalences, what we want to capture is the formalisation of JaLAPI. While there exist formalisations of the Java language and the JVM they are too low-level to be useful to us for our purposes here. In order to demonstrate the ability of JaLAPI to implement LySa models we present reduction rules for the JaLAPI methods.

The rules take the standard form of requirements on the top, code and outcome on the bottom. In order to accurately describe the scenario we have four semantic objects, a context, a state, a network and a keystore. The context, Γ, is a list of type assertions that describes the type of various objects in the scenario. To say that an object m has type *Message* in the scenario the requirement reads, $\Gamma \vdash \{m \mapsto Message\}$. If a line of Java creates a new object, $m2$ with the type *Message* we can build a new context $\Gamma \uplus \{m2 \mapsto Message\}$. We have a simple typing hierarchy used to encapsulate the relationships between messages and their contents.



Another semantic object we deal with is the state of the objects, S, which records what their content is at a particular time, as a simple mapping. To say that a *Message* m has content (c) which is a triple with three elements, w, x and y, we would say $S(m) = (c : \{w,x,y\})$. If a rule makes a change to the state of an object, for example adding a further element z to the end of this message the new state which results from this is $S[m := c : \{w,x,y,z\}]$. Simultaneous updates to the state are written as $S[v := w,\ m := c : \{x,y,z\}]$. The previous example demonstrates the difference in discussing *MsgPart* and *Message* objects. With *MsgPart* objects the state of the objects is given as $v := w$. With a *Message* object we give the contents of the message as a list like $m := c : \{x,y,z\}$.

When discussing communication we also need to discuss the contents of a Network, N. When messages are transmitted over the network extra fields are needed

for source and destination, thus adding a new message to the network looks like $N@(s:A, d:B, c:\{x,y,z\})$.

The final semantic object we have is a keystore, KS. The syntax for adding and retrieving keys is the same as the syntax for the context thus to add a new key with the label K we write $KS \uplus \{(k,K)\}$ and to check that there is a key with that label $KS \vdash \{(k,K)\}$.

When the action a particular rule is formalising does not affect one of these semantic objects we do not include that semantic object in the rule. This simplifies the rule and additionally allows you, the reader, to easily see the effect of the action.

7.2.1 `KeyGeneration` Class

We now present formal rules of the methods presented in Section 5.3. The first class we introduced was the `KeyGeneration` class which featured methods for generating either a `SecretKey` or a `KeyPair`. We shall start by examining the method for generating a `SecretKey`, named `generateSharedKey`. This method creates a new object and initialises the state of this object although it does not involve either the keystore or the Network. We will thus concern ourselves exclusively with the semantic objects Γ and S.

We use the preconditions of this rule to denote that the method `generateSharedKey` constructs a `Key`. The result of this rule is that the context is appended with a new object of type *Key*. The state of this object is also set signifying that the content of this *Key* is a key generated by the method `generateSharedKey`. We summarise this as follows.

CREATEKEY

$$\frac{\mathit{generateSharedKey}() = k}{(\Gamma, S) \vdash \texttt{SecretKey key} = \texttt{generateSharedKey}() \xrightarrow[\nu K\ \mathit{key}]{} (\Gamma \uplus \{\mathit{key} \mapsto \mathit{Key}\}, S[\mathit{key} := k])}$$

For the method `generateKeyPair` we note that this method generates a tuple of keys, K+ and K-. Much like CREATEKEY the context and state are updated to reflect the new object being of type *KeyPair* and consisting of a tuple of accompanying keys.

CREATEKEYPAIR

$$\frac{\mathit{generateKeyPair}(1024) = \{K+, K-\}}{(\Gamma, S) \vdash \texttt{KeyPair keypair} = \texttt{generateKeyPair}(1024) \xrightarrow[\nu K\pm\ \mathit{keypair}]{} \begin{array}{l}(\Gamma \uplus \{\mathit{keypair} \mapsto \mathit{KeyPair}\},\\ S[\mathit{keypair} := \{K+, K-\}])\end{array}}$$

We now have to provide formalisations for method calls that are not part of JaLAPI but are a part of Java's built in crypto functions. These are the methods for extracting the public and private keys from a *KeyPair* object. For this method call to succeed there needs to be an already generated *KeyPair* object which consists of a pair of keys. After the method invocation the context must be updated to reflect the creation of a new key object. Depending on whether the `getPublic` or `getPrivate` method is called, the state of this new key is either the public or private key from the *KeyPair* tuple. This can be more formally expressed in the two rules below.

CREATEPRIVATEKEY

$$\frac{\Gamma \vdash \{keypair \mapsto KeyPair\} \qquad S(keypair) = \{\_, x\}}{(\Gamma, S) \vdash \texttt{PrivateKey key} = \texttt{keypair.getPrivate()} \xrightarrow[key=keypair-]{} (\Gamma \uplus \{key \mapsto Key\}, S[key := x])}$$

CREATEPUBLICKEY

$$\frac{\Gamma \vdash \{keypair \mapsto KeyPair\} \qquad S(keypair) = \{x, \_\}}{(\Gamma) \vdash \texttt{PublicKey key} = \texttt{keypair.getPublic()} \xrightarrow[key=keypair+]{} (\Gamma \uplus \{key \mapsto Key\}, S[key := x])}$$

7.2.2 `Message` Class

We now examine the rules for creating and populating a message. This task has been made easier by using a bespoke `Message` object as we can restrict the methods of accessing the content of the message. The rule for the `Message` constructor is given below. It has no preconditions and creates a new object in the same manner as the rules in the previous section. We also see the first use of the formal specification of the *Message* structure as we set the content of the message to the empty set.

NEWMESSAGE

$$\frac{}{(\Gamma, S) \vdash \texttt{Message m} = \texttt{new Message()} \xrightarrow[\nu M\ m]{} (\Gamma \uplus \{m \mapsto Message\}, S[m := c : \{\emptyset\}])}$$

The next rule we shall examine adds elements to a *Message* object. Unlike the previous rule, there are certain requirements and assumptions as to the situation before the method call. The types of the objects are very important. The object we are trying to add must be of type *MsgPart* as described in the above diagram. Equally the object we are trying to add the message part to must be a *Message* object. As well as these requirements on the context we must also make some assumptions about the state of the message before we add any additional parts so that we can show the result of this action.

In this instance the context does not change but we append the new message part to the end of the tuple showing the contents of the *Message*. To show this we present the tuple as only having one element prior to the method call and an additional one with the name of the variable we are adding afterwards.

MESSAGEADD

$$\frac{\Gamma \vdash \{m \mapsto Message, v \mapsto MsgPart\} \qquad S(m) = (c : \{x\})}{(\Gamma, S) \vdash \texttt{m.add(v)} \xrightarrow[m+=v]{} (\Gamma, S[m := c : \{x, v\}])}$$

We shall cover the methods which deal with accessing the *Message* in Section 7.2.4 where they are used along with the `check` method.

7.2.3 `Network` Class

Although the JaLAPI Network class as described in Section 5.3.4 has multiple methods, the only one that has an observable effect in LySa is the `send` method.

We use the formalisation of the Network semantic object introduced at the start of Section 7.2 for the first time in this formalisation. In order to send a message we need the source. This does not come from the method call but from the Network `register` method invocations in the set-up class. We represent this here in the name of the message rule with 'A' as a placeholder for the current principal and used as the source attribute for the message in the following rule. The other preconditions that must be satisfied for this rule are that the types of the objects are correct. We also say that the message has a given content. As a consequence of this rule the Network is updated to feature a message with the destination from the method call and the content taken from the content of the *Message* object being added.

SEND(A)

$$\frac{\Gamma \vdash \{m \mapsto Message, net \mapsto Network\} \qquad S(m) = (c : \{x, y, z\})}{(\Gamma, N, S) \vdash \texttt{net.send(B,m)} \xrightarrow[m \to B]{} (\Gamma, N@(s : A, d : B, c : \{x, y, z\}), S)}$$

7.2.4 `Principal` Class

The corresponding receive process to the just introduced send is not a method invocation but the method header for the `processIncoming` method. The following rule specifies the inverse of the above send and removes a message from the Network and adds it to the context and state including having source and destination attributes set.

RECEIVE(B)

$$\frac{N' = (s : A, d : B, c : \{x,y,z\})@N}{(\Gamma, N', S) \vdash \texttt{processIncoming(Message m)} \xrightarrow[A \to m]{} \begin{array}{c}(\Gamma \uplus \{m \mapsto Message\},\\ N, S[m := s : A, d : B, c : \{x,y,z\}])\end{array}}$$

The next three rules are formalisations of variations on the `check` method using the `getSource`, `getDest` and `getNext` methods from the `Message` class. Rather than present separate rules for the check and get methods we simplify the situation by combining these rules in the manner they will actually be used. The CHECKSOURCE and CHECKDEST rules are nearly identical. The precondition for these rules state that whichever attribute we are checking has the same value as the second parameter of the check method.

CHECKSOURCE

$$\frac{\Gamma \vdash \{m \mapsto Message\} \qquad S(m) = (s : A)}{(\Gamma, S) \vdash \texttt{check(m.getSource(), A)} \xrightarrow[A == m.s]{} (\Gamma, S)}$$

CHECKDEST

$$\frac{\Gamma \vdash \{m \mapsto Message\} \qquad S(m) = (d : B)}{(\Gamma, S) \vdash \texttt{check(m.getDest(), B)} \xrightarrow[B == m.d]{} (\Gamma, S)}$$

The difference between these rules and the CheckPart rule below is that the content of the Message is modified when checking individual message parts so that subsequent method calls will not be accessing the same message part.

CHECKPART

$$\frac{\Gamma \vdash \{m \mapsto Message\} \qquad S(m) = (c : \{x,y,z\})}{(\Gamma, S) \vdash \texttt{check(m.getNext(), x)} \xrightarrow[x == m--]{} (\Gamma, S[m := c : \{y,z\}])}$$

The `getNext` method is used again for the second part of receiving a message; variable instantiation. Although this does not involve any methods from the JaLAPI principal class, we feature it in this section as it fits here better than in the Message class. Like the previous CHECKPART rule, we state that as a consequence the message's contents are modified, In this instance the removed message part is stored in a new variable.

VARIABLEASSIGNMENT

$$\frac{\Gamma \vdash \{m \mapsto Message\} \qquad S(m) = (c : \{x,y,z\})}{(\Gamma, S) \vdash \texttt{String i = m.getNext()} \xrightarrow[i == m--]{} (\Gamma \uplus \{i \mapsto MsgPart\}, S[m := c : \{y,z\}, i := x])}$$

At this point we have presented rules for composing, sending, receiving and processing a message as well as generating keys. We now present the rules for using these keys, starting with usage of the keystore, sending and receiving keys finishing with formalisations of encryption and decryption. Elyjah and Hajyle both require keys to be registered with a keystore so that there is a single name that can be used to identify keys instead of using multiple variable names. We now present the rules for registering and retrieving keys from this keystore. In order to register a key the object we are attempting to store must be validated as a Key object. After running the method, the keystore is appended with an additional tuple containing the aforementioned key and a label.

REGISTERKEY

$$\frac{\Gamma \vdash \{key \mapsto Key\}}{(\Gamma, KS) \vdash \texttt{registerKey(key,"K")} \xrightarrow[KS \uplus (key,K)]{} (\Gamma, KS \uplus \{(key,K)\})}$$

We also have the accompanying method for retrieving a key from the keystore. In this instance the preconditions state that the keystore must contain a tuple containing a key and a label.

GETKEY

$$\frac{KS \vdash \{(k,KAB)\}}{(\Gamma, KS) \vdash \texttt{SecretKey key} = \texttt{(SecretKey)keys.get("KAB")} \xrightarrow[KS \to (key,KAB)]{} (\Gamma \uplus \{key \to Key\}, S[key := k])}$$

When sending a key the `sendKey` method can be used in two different ways. These reflect the options in the *m.add* V_i diagram in Section 7.1.2.1. Much like the labels for the transitions we use the notation $K(x)$ to denote that x is a *Key* in a *String* representation.

SENDKEY

$$\frac{\Gamma \vdash \{m \mapsto Message, key \mapsto Key\} \qquad S(m) = (c : \{x\})}{(\Gamma, S) \vdash \texttt{m.add(sendKey(key))} \xrightarrow[m+=K(key)]{} (\Gamma, S[m := (c : \{x, K(key)\})])}$$

SENDKEY II

$$\frac{\Gamma \vdash \{key \mapsto Key\}}{(\Gamma, S) \vdash \texttt{String k} = \texttt{sendKey(key)} \xrightarrow[k=K(key)]{} (\Gamma \uplus \{k \to String\}, S[k := K(key)])}$$

As for receiving a message, due to the different algorithms used, there are different methods for each key type. The rules presented below show the rules for all three types, `SecretKey`, `PublicKey` and `PrivateKey`. In all of these rules a *String* representation of a key is the first element in a *Message*. After the method is called, a new *Key* subtype exists whose state is the key in normal form.

RECEIVESECRETKEY

$$\frac{\Gamma \vdash \{m \mapsto Message\} \qquad S(m) = (c : \{K(x), y, z\})}{(\Gamma, S) \vdash \texttt{SecretKey k} = \texttt{receiveSecretKey(m.getNext())} \xrightarrow[K(k)=m--]{} \begin{array}{c}(\Gamma \uplus \{k \to Key\},\\ S[k := x],\\ S[m := (c : \{y, z\})])\end{array}}$$

RECEIVEPUBLICKEY

$$\frac{\Gamma \vdash \{m \mapsto Message\} \qquad S(m) = (c : \{K(x), y, z\})}{(\Gamma, S) \vdash \texttt{PublicKey k} = \texttt{receivePublicKey(m.getNext())} \xrightarrow[K(k)=m--]{} \begin{array}{c}(\Gamma \uplus \{k \to Key\},\\ S[k := x],\\ S[m := (c : \{y, z\})])\end{array}}$$

RECEIVEPRIVATEKEY

$$\frac{\Gamma \vdash \{m \mapsto Message\} \qquad S(m) = (c : \{K(x), y, z\})}{(\Gamma, S) \vdash \texttt{PrivateKey k} = \texttt{receivePrivateKey(m.getNext())} \xrightarrow[K(k)=m--]{} \begin{array}{c}(\Gamma \uplus \{k \to Key\},\\ S[k := x],\\ S[m := (c : \{y, z\})])\end{array}}$$

For symmetric encryption and decryption the rules are mirror images of each other. For encryption the preconditions state that there must be a Message with certain content and afterwards there is an *EncMessage* with the same content but encrypted with a previously defined *Key*. The inverse is that there is an *EncMessage* with content encrypted with a Key that generates a *Message* object with the same content but not encrypted.

ENCRYPT

$$\frac{\Gamma \vdash \{m \mapsto Message, K \mapsto Key\} \qquad S(m) = (c : \{x, y, z\})}{(\Gamma, S) \vdash \texttt{String enc} = \texttt{encrypt(m,K)} \xrightarrow[enc=\{m\}_K]{} (\Gamma \uplus \{enc \mapsto EncMessage\}, S[enc := \{x, y, z\}_K])}$$

DECRYPT

$$\frac{\Gamma \vdash \{K \mapsto Key, enc \mapsto EncMessage\} \qquad S(enc) = (c : \{x, y, z\}_K)}{(\Gamma, S) \vdash \texttt{Message m} = \texttt{decrypt(enc,K)} \xrightarrow[m=dec(\{m\}_K)]{} (\Gamma \uplus \{m \mapsto Message\}, S[m := c : \{x, y, z\}])}$$

Hashing is modelled in LySa by using asymmetrical encryption where the accompanying decryption key does not exist. The formalisation of the JaLAPI hash method is provided below. Rather than the hashed block be modelled as an *EncMessage* a *String* is used as this message should not be able to be decrypted.

HASH

$$\frac{\Gamma \vdash \{msg \mapsto Message\} \qquad S(msg) = (c : \{x, y, z\})}{(\Gamma, S) \vdash \texttt{String h} = \texttt{hash(msg)} \xrightarrow[h=\{msg\}_{MD5+}]{} (\Gamma \uplus \{h \mapsto String\}, S[h := \{x, y, z\}_{MD5+}])}$$

As an encrypted block will typically be added to another *Message* object to be sent to another principal, there presents another scenario where the `encrypt` method is used directly in an `add` method invocation.

ENCRYPT II

$$\frac{\Gamma \vdash \{m \mapsto Message, m2 \mapsto Message, K \mapsto Key\} \qquad S(m) = (c : \{x,y,z\}) \qquad S(m2) = (c : \{a,b,c\})}{(\Gamma, S) \vdash \texttt{m.add(encrypt(m2,K))} \xrightarrow[m+=\{m2\}_K]{} (\Gamma, S[m := \{x,y,z,\{a,b,c\}_K\}])}$$

Asymmetric encryption has exactly the same rule as symmetric encryption as it does not matter whether a public, private or shared key is used. On the other hand when one key is used for asymmetric encryption the other must be used for decryption.

ADECRYPT

$$\frac{\begin{array}{lll}\Gamma \vdash & \{K \mapsto Key,\ enc \mapsto EncMessage & S(enc) = (c : \{x,y,z\}_{K+}) \\ & keypair \mapsto KeyPair\} & S(keypair) = \{K+, K-\}\end{array}}{(\Gamma, S) \vdash \texttt{Message m} = \texttt{decrypt(enc,K-)} \xrightarrow[m=dec(\{m\}_{K-})]{} (\Gamma \uplus \{m \mapsto Message\}, S[m := c : \{x,y,z\}])}$$

ASIGNATURE

$$\frac{\begin{array}{lll}\Gamma \vdash & \{K \mapsto Key, enc \mapsto EncMessage & S(enc) = (c : \{x,y,z\}_{K-}) \\ & keypair \mapsto KeyPair\} & S(keypair) = \{K+, K-\}\end{array}}{(\Gamma, S) \vdash \texttt{Message m} = \texttt{decrypt(enc,K+)} \xrightarrow[dec(m=\{m\}_{K+})]{} (\Gamma \uplus \{m \mapsto Message\}, S[m := c : \{x,y,z\}])}$$

We have two more methods for generating message parts. The first, `generateMessage` is used only to replicate LySa's ν process for message parts. The second `generateNonce` has more functionality and sets the new variable as a freshly generated nonce.

NEWSTRING

$$\frac{}{(\Gamma, S) \vdash \texttt{String msg} = \texttt{generateMessage}(\text{“}\texttt{Message}\text{”}) \xrightarrow[\nu S\ msg]{} \begin{array}{l}(\Gamma \uplus \{msg \mapsto String\}, \\ S[msg : \text{“}Message\text{”}])\end{array}}$$

NEWNONCE

$$\frac{generateNonce() = n}{(\Gamma, S) \vdash \texttt{String nonce} = \texttt{generateNonce}() \xrightarrow[\nu N\ nonce]{} (\Gamma \uplus \{nonce \mapsto String\}, S[nonce : n])}$$

As a summary of which labels and rules relate to which method call we provide the table in Table 7.2.4.

7.3 Case study

To show how these rules all fit together we will look at a very simple protocol and see how the LySa and Java model and implement it. The standard narration is as follows:

Rule Name	Label	Java Code
CREATEKEY	$\nu K\ key$	`SecretKey key = generateSharedKey()`
CREATEKEYPAIR	$\nu K \pm\ keypair$	`KeyPair keypair = generateKeyPair(1024)`
CREATEPRIVATEKEY	$key = keypair-$	`PrivateKey key = keypair.getPrivate()`
CREATEPUBLICKEY	$key = keypair+$	`PublicKey key = keypair.getPublic()`
NEWMESSAGE	$\nu M\ m$	`Message m = new Message()`
MESSAGEADD	$m+ = v$	`m.add(v)`
SEND(A)	$m \rightarrow B$	`net.send(B, m)`
RECEIVE(B)	$A \rightarrow m$	`processIncoming(Message m)`
CHECKSOURCE	$A == m.S$	`check(m.getSource(),A)`
CHECKDEST	$A == m.D$	`check(m.getDest(), A)`
CHECKPART	$x == m--$	`check(m.getNext(), A)`
VARIABLEASSIGNMENT	$i = m--$	`String i = m.getNext()`
REGISTERKEY	$KS \uplus (key, K)$	`registerKey(key, "K")`
GETKEY	$KS \rightarrow (key, K)$	`SecretKey key = (SecretKey)keys.get("K")`
SENDKEY	$m+ = K(key)$	`m.add(sendKey(key))`
SENDKEY II	$k = K(key)$	`String k = sendKey(key)`
RECEIVESECRETKEY	$K(k) = m--$	`SecretKey k = receiveSecretKey(m.getNext())`
RECEIVEPUBLICKEY	$K(k) = m--$	`PublicKey k = receivePublicKey(m.getNext())`
RECEIVEPRIVATEKEY	$K(k) = m--$	`PrivateKey k = receivePrivateKey(m.getNext())`
ENCRYPT	$enc = \{m\}_k$	`String enc = encrypt(m,k)`
ENCRYPT II	$m+ = \{m2\}_k$	`m.add(encrypt(m2,k))`
HASH	$h = \{msg\}_{MD5+}$	`String h = hash(msg)`
DECRYPT	$m = dec(\{enc\}_k)$	`Message m = decrypt(enc,k)`
ADECRYPT	$m = dec(\{enc\}_{k-})$	`Message m = decrypt(enc,k-)`
ASIGNATURE	$m = dec(\{enc\}_{k+})$	`Message m = decrypt(enc,k+)`
NEWSTRING	$\nu S\ msg$	`String msg = generateMessage("msg")`
NEWNONCE	$\nu N\ nonce$	`String nonce = generateNonce()`

Table 7.1: Summary of rules,labels and names

$$A \rightarrow B : K+$$
$$B \rightarrow A : \{msg\}_{K+}$$

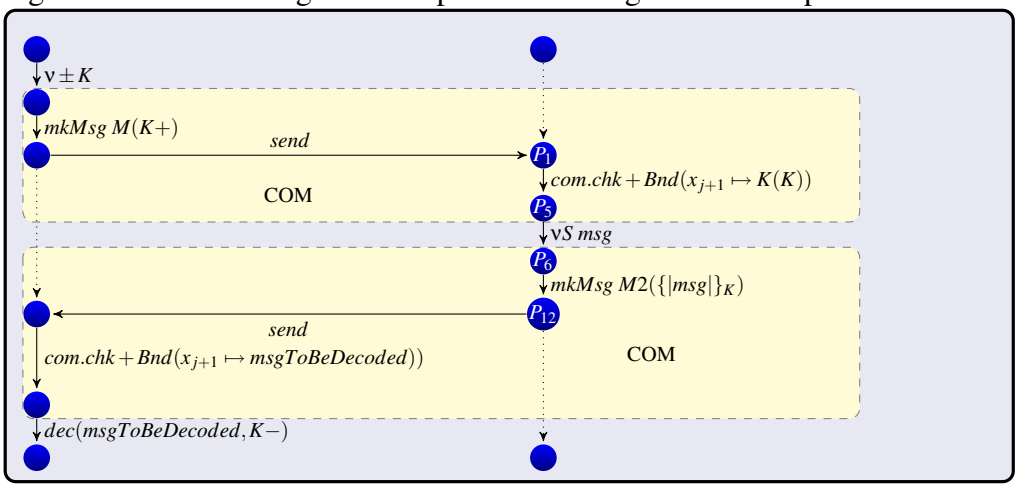
The clear goal of this protocol is to allow secure communication between principals A and B by having B's message to A being encrypted. While this protocol has many security flaws it is sufficient to demonstrate key generation, public key encryption and decryption as well as communication between principals. The LySa code for this protocol is:

$(\nu \pm K)\langle A,B,K+\rangle.(B,A;msgToBeDecoded).$
decrypt msgToBeDecoded as $\{|;msg|\}_{K-}$ [at a1 orig { b1 }] in 0
|
$(\nu$ message$)$ (A,B;K).$\langle B,A,\{|message|\}_K$ [at b1 dest { a1 }] $\rangle.0$

This protocol can be broken up into the various rules we have presented in this chapter. The first step is the creation of a new keypair. This process is described in both the LySa reduction rule ANEW, and the accompanying rules presented in Section 7.2.1. The next part of the protocol is sending a message from principal A to principal B. This is covered in the LySa reduction COM rule but for the Java reduction rules this has to be broken down in the following generalised steps. Principal A creates a new message object and adds the public key from the previously generated key pair to it. This process is formally described in the MKMSG and ADD diagrams. The message is then sent to principal B. Principal B receives this message and checks the first two elements in the message are the expected source and destination, this is detailed in the COM.CHK section. It then binds the third element to a new variable, *x*, taking the incoming `String` and transforming it into a `Key` object as pictured in the BNDMSG diagram. Additionally a new `String`, *msg*, is created, formally described in the LySa NEW rule which for reasons of brevity is left out of this paper. Principal B then creates a new `Message` and adds to it the string representation of a new `Message`, which has the previously created `String` *msg* added to it, encrypted with the variable *x*, which was the Key sent by A. This message is sent to the protocol initiator, A, who checks the first two elements are the expected source and destination and stores the third element in a variable. This is the `String` representation of the encrypted message. A retrieves the key pair's private key and uses this to decrypt the message as detailed in the DEC picture and rules.

The next page shows how the Java program for principal B is represented as a tree of operational rules which match up with the following diagram showing a complete

overview of the protocol. This demonstrates how the rules which correspond to LySa processes expand and eventually terminate with Java statements using JaLAPI. This Java code is extracted from the protocol implementation generated by running Hajyle on the LySa code at the start of this section. Below is an overview of the entire protocol in diagrammatic form using the rules provided throughout this chapter as shorthand.



In this section we have shown how a block of Java code relates to JaLAPI rules leading to a complete transition system for the protocol. In the diagram above, the typically unnamed nodes have been labelled to denote the states within the transition system used on the next page which has been additionally recapped below.

$$(\Gamma, S) \vdash P_1 \xrightarrow{com.chk+bnd_{x_{j+1}\to K(k)}} P_5 \xrightarrow{\nu S\ msg} P_6 \xrightarrow{mkMsg\ M2(\{|msg|\}_K)} (\Gamma', S') \vdash P_{12}$$

The Java program J represented by this transition system should then preserve the protocol narration and security properties of the original LySa model L according to the $\approx$ relation as defined at the start of this chapter.

$$
\begin{array}{l}
(\Gamma, S) \vdash P_1 \xrightarrow{com.chk+bnd_{x_{j+1}-K(k)}} P_5 \xrightarrow{vS\ msg} P_6 \xrightarrow{mkMsg\ M2(\{|msg|\}_K)} (\Gamma', S') \vdash P_{12} \\
\quad P_1 \xrightarrow{com.chk+Bnd_{x_{j+1}-K(k)}} P_5 \quad P_5 \xrightarrow{vS\ msg} P_6 \quad P_6 \xrightarrow{vM\ m2} P_7 \quad P_7 \xrightarrow{mkMsg\ mEnc} P_9 \quad P_9 \xrightarrow{Enc(K)\ msg} P_{11} \quad P_{11} \xrightarrow{add\{|msg|\}_K} P_{12} \\
\quad P_7 \xrightarrow{vM\ mEnc} P_8 \quad P_8 \xrightarrow{add\ msg} P_7 \quad P_{11} \xrightarrow{\{m\}_K} P_{10} \quad P_{10} \xrightarrow{KS+(key2,K)} P_{12} \\
\quad P_1 \xrightarrow{com.chk\ m} P_3 \quad P_3 \xrightarrow{bndMsg_{x_{j+1}-K(k)}} P_5 \quad P_3 \xrightarrow{bnd\ x_{j+1}-K(k)} P_5 \quad P_3 \xrightarrow{K(key)=m--} P_4 \quad P_1 \xrightarrow{A==S(m)} P_2 \quad P_2 \xrightarrow{B==D(m)} P_3 \quad P_5 \xrightarrow{KS+(key,K)} P_5 \quad P_3 \xrightarrow{m2+=\{|msg|\}_K} P_{12} \\
\quad P_1 \quad P_5 \quad P_6 \quad P_7 \quad P_7 \quad P_9 \quad P_{11} \quad P_{12} \quad P_7 \quad P_8 \quad P_{10} \quad P_{12} \quad P_1 \quad P_3 \quad P_5 \quad P_3 \quad P_4 \quad P_2 \quad P_3 \quad P_5 \quad P_5 \quad P_3 \quad P_{12} \\
\quad \text{check(m.getSource(),A)} \quad \text{check(m.getDest(),B)} \quad \text{PublicKey key = receivePublicKey(m.getNext())} \quad \text{registerKey(key,"K")} \\
\quad \text{String msg = generateMessage("message")} \quad \text{Message m2 = new Message()} \quad \text{Message mEnc = new Message()} \quad \text{mEnc.add(msg)} \\
\quad \text{PublicKey key2 = (PublicKey)keys.get("K")} \quad \text{String enc = encrypt(mEnc, key2, b1, a1)} \quad \text{m2.add(enc)}
\end{array}
$$

Chapter 8

Conclusions

In this final chapter the work presented in this dissertation is summarised. To put this work into context, related work is discussed in Section 8.1. Like the later sections we divide the work into three distinct areas. LyTE and PAMeLA are discussed separately from each other and Elyjah and Hajyle. Possible extensions to all of these distinct sections are provided in Section 8.2. Finally, to conclude the chapter and the dissertation as a whole, a final discussion of the functionality of the work and the perceived larger value is presented in Section 8.4.

8.1 Related Work

8.1.1 Work Related to LyTE

Examples of applications which provide visual representations of cryptographic protocols are ProtoViz[32], GRASP[72], GRACE[25] and TECP[81]. These tools are primarily designed as education packages for teaching students about cryptographic protocols. While LyTE would be more suitable for teaching students about process calculi such as LySa, it could still be used to demonstrate cryptographic protocols by a demonstrator familiar with LySa. Additionally, these other tools use their own language for describing a protocol whereas LyTE uses the LySa language so the protocols can be analysed for security flaws, which is an important point when teaching cryptographic protocols.

The LyTE toolkit provided also owes a debt of inspiration to the PEPA Eclipse Plugin [78] for some of the syntax highlighting and LyTeX ideas.

8.1.2 Work Related to PAMeLa

A theoretical approach to performance evaluation for security protocols has already been attempted in work such as [15]. This work relies on establishing a new extended operational semantic for LySa in which each transition is assigned a label. Here we provide an easy-to-use alternative that fits in with both the PEPA Eclipse Plug-in and the LySa Toolkit in Eclipse (LyTE).

There is related work on performance modelling of concurrent systems [8] or non-security protocols[27, 38] however the addition of cryptographic operations in this work means there is significant separation between the analysis provided by these works. As the input for PAMeLa is automatically generated from a LySa model no extra knowledge or input language training is needed.

8.1.3 Work Related to Elyjah and Hajyle

There are many alternate methods for modelling and analysing cryptographic protocols such as [22, 1, 63, 73, 49]. Equally, not all attempts at increasing program security depend on cryptography. The work on proof-carrying code[52, 4] attaches machine-checkable proofs of program properties instead of cryptographic certificates.

There are several examples of work related to Hajyle, where implementations of protocols are generated from specifications or formal models. The following is not an exhaustive list, merely those judged to be most closely related or inspirational to the development of Hajyle.

COSP-J[28] is a protocol compiler that generates Java implementations of code from an input script based on Casper[46]. In [41] a tool, ACG-C#, is detailed where C# code is generated from a modified Casper script that is translated into CSP[40] and then verified with the FDR tool[69]. The Java and C# that is generated has similar API method calls to JaLAPI which suggests that our method is reasonable. A downside of both of these tools is that the input is a slightly modified version of the Casper script for which there is no automatic translation tool. This additional step seems to be a potential source of errors that needs to be addressed.

The SPEARII framework first described in [70] details the reasons for providing source code generation as part of a complete protocol suite. This was expanded in [47]. The Protocol Code Generator takes information from two data specifications in the SPEAR framework however all of this information is input via a graphical user interface that appears to reduce the learning curve of the system. The generated code

is also in the Java language.

Java code is also generated in work by Muller and Millen[48]. In this instance the source language is the CAPSL intermediate language CIL. While this may appear to be a problem regarding the equivalence of the protocol expressed in two languages as protocol analysis is performed on the CIL representation there is no such problem. The only way a problem may develop is if someone else was using a CAPSL representation of a protocol to create another version of the protocol for interoperability reasons and missed some of the vital information only contained in the CIL version.

Spi[1] is the source language for Spi2Java[67, 77, 64] which as the name suggests translates into Java implementations. This is one of the more developed tools in the field and is no doubt the inspiration for tools such as Expi2Java[23] and Spi2F#[76]. This work is most closely related to Hajyle as the Spi language is closely related to LySa and the produced Java shares similar method calls. [64] contains a more complete formalisation than we present in Chapter 7.

A different approach is taken in [74]. This work presents a toolkit named *AGVI*, *Automatic Generation*, *Verification and Implementation of Security Protocols* where a protocol designer inputs the specifications and requirements of a given scenario and the toolkit generates several protocols that fit these requirements. These protocols can then be checked and implemented in Java. While this is an intriguing option particularly for those with zero experience with cryptographic protocols the ability to generate traditional protocols for interoperability reasons is a necessary and powerful addition.

While analysing cryptographic protocols directly from source code seems to be fairly new, there is other interesting work in the area. Jan Jürjens has written papers detailing work on analysing Java [43] and C [44] implementations of cryptographic protocols. This work relies on the developer adding annotations to the code which are then validated against the program behaviour by run-time analysis. This makes it ideally suited to analysing legacy code. Elyjah analyses the source code itself resulting in less extra work for a developer creating a new implementation, however this method does restrict Elyjah's ability to analyse arbitrary code.

There is only a limited amount of directly related work in the area of code-to-model verification. Relevant works are Goubault-Larrecq and Parrennest's work [36] on analysis on C code, which does not attempt to deduce cryptographic protocols from the code but does analyse the reachability properties of the code. Also relevant is work [66] by Poll and Schubert where they study an implementation of SSH in Java using the specification language JML. The most closely related work is the Microsoft

Research tool FS2PV [9] which translates F# programs to a low-level π calculus model. This can then be analysed by ProVerif to provide security analysis of the implementation. The follow-up to this work [7] uses typing rules and dramatically reduces the time needed for verification. The requirement in this work for special refinement types is comparable to Elyjah requiring the use of a special API. In addition F# is not used as much as the Java language that Elyjah is geared towards. This work has been shown to work on larger examples than Elyjah has so far been subject to although the time taken for protocols such as the Otway-Rees is close to two minutes whereas the static analysis that Elyjah and the LySatool use means this analysis is less than a second.

ASPIER[26] represents impressive progress on analysing C code, however run times are even higher than F7. The referenced paper gives an analysis time of over 4 and a half days for analysing secrecy properties in a protocol. Analysis of authentication takes additional time above this.

Irrespective of any technical differences, the work presented in this dissertation has two clear advantages over this related work. Having complementary tools for both model-to-code and code-to-model nullifies many of the arguments as to which direction is superior. The more developed field of code generation has a major drawback in that the developer cannot be certain that any changes made preserve the security properties of the original model. This means the implementation can only be said to be based on a verified model and the verification cannot be used as part of any service level agreement. While implementation analysis is starting to become a hot-topic most of the work, including Elyjah, requires developers to do more work than they would if they were not planning on analysing their work. This may include adding additional annotations, using an unfamiliar API or even an unfamiliar language. Being able to generate code from a model actually saves developers time rather than using up more of theirs. This is powerful evidence is support of using a specialised language such as LySa for modelling the protocol over a more general form of model checking such as Horn clauses from which it is harder to recreate a protocol implementation.

Another advantage of the work presented here is that it is integrated into the Eclipse Development Environment and analysis results can be achieved in a one-step process so protocol analysis is more likely to be used by developers. If we are to note that formal methods are currently underused by developers then we must expect that developers are equally unlikely to use any of these tools unless every conceivable effort is made to make using such tools as natural as possible.

8.2 Future Work

The most valuable extension to this work would be to achieve true model driven development with Elyjah and Hajyle. Certain information is lost in translation from Java to LySa, it would be beneficial if this information was restored if and when this LySa model was used by Hajyle to generate code where possible. Ideally as much as possible of this implementation would be kept despite changes to the LySa model.

JaLAPI can certainly be expanded with more API functions, perhaps including nonce manipulation and even choice which would then need integration into the two tools Elyjah and Hajyle. The syntax for incoming message switching could also be examined, although this would mostly be to make formalisation simpler and to find a way to enforce the correct syntax.

A feature of the API that it supports multiple implementations could be exploited to include different versions of implementations for interoperability with other versions of the same protocol. In support of such extended implementations, further work could also include the development of tools to check the API implementations are sound.

There are additional tools that could be added to the LySa Toolkit in Eclipse. We believe the most interesting additions would be LySa model generation from Alice-Bob style narration as described in [18, 24]; automatic deduction of crypto-points from a LySa model; equivalence testing between multiple LySa models and automatic addition, removal of additional LySa annotations for analysis of LySa models as in [34].

Automated LySa generation would be useful as it would allow a user to generate a Java implementation based on the typical Alice and Bob narration that is often used to describe protocols. While the generated LySa would still have to be verified and validated by tools such as the LySatool and AnaLySa as the specification or translation may not be correct, the process would be a fast way of generating skeleton code which could still be checked even after making necessary changes. The challenge in this process is that protocol narrations miss out a lot of data as mentioned in Section 2.2.1. This information has to be inferred from the narrations or provided in supporting materials to determine what information is known prior to the start of a protocol run and what actions a recipient of a message undertakes upon receiving a message. Protocol narrations do not specify whether a message part is checked, assigned to a variable or decrypted, and if so, with what key. Additionally, information about the deployment scenario is needed to give greater clarity to the role a principal plays, for example the name S usually refers to a fixed authentication server which is important information

if constructing a Meta-LySa model.

Currently the assertions that the crypto-points represent are the only part of the code which is not a natural part of the implementations. They represent annotations in a system where we have tried to minimise extra information required to perform translations. Having crypto-points automatically deduced would mean no assertions are required in the Java code.

LySa equivalence would be useful to see if a program represents the same LySa model before and after modification without comparing LySa models or analysis results by hand. This could additionally be useful for interoperability reasons.

For PAMeLa the most important improvement that can be made is to improve accuracy of the rates generated by the AutoRate tool in Section 4.5. Due to the resources available it was not possible to gather rates from multiple device types and across different communication mediums. The rates we have provided are based on measuring the time taken to perform tasks on one computer and then scaling these to other devices using processor chip speed as a guide. However, we have provided a framework that allows the rates we have provided to be edited by a user. The Eclipse Preferences option provides a persistent method of storing preferences over multiple workspace sessions. At any point the user can also choose to revert their options to the default rates. Figure 8.1 shows the preference page for the AutoRate tool.

8.3 Use Cases

8.3.1 Protocol Designer

One potential user of this series of tools would be a protocol designer. This person is more familiar with process calculi than programming languages such as Java. They would be expected to start with an empty LySa file and use the LySa Toolkit in Eclipse to assist them in creating a LySa model. At the early stage, elements of the LySa Editor would be of most use, namely features such as syntax highlighting and syntax checking. Once a first draft of the model has been completed, tools such as the AnaLySa and VisuaLySa can then be used by the designer to examine how the protocol operates, checking that message parts are being sent correctly and the protocol terminates as expected. These tools also reveal information about the LySatool results but our protocol designer is likely to prefer the traditional LySatool results as provided through LyTE. They will be familiar enough with these results to diagnose any problems and fix them

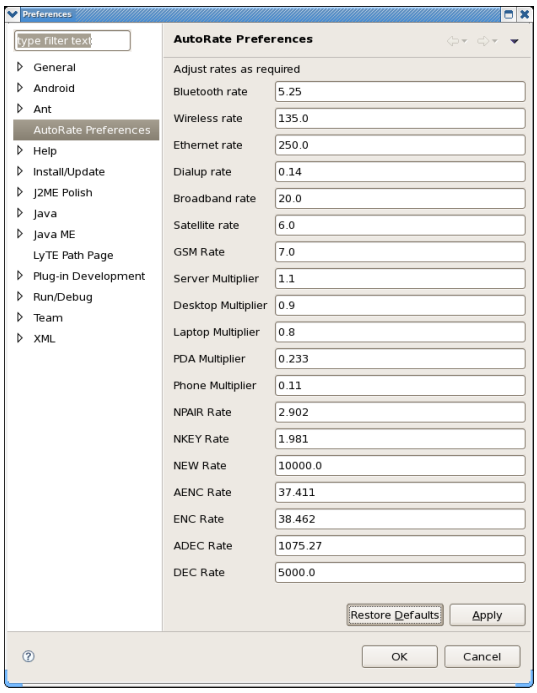


Figure 8.1: AutoRate Preferences Page

directly.

If they have a particular deployment scenario in mind they may choose to use PAMeLa to examine how suitable their new protocol is using the default rates provided by the tools.

It is also possible they may wish to dabble in implementations and use Hajyle to generate Java such that can add some basic logging information to check that encryption and decryption is achieved successfully for the confidential payload. By running the generated Java program they designer can check that the newly-developed protocol is live as well as safe.

8.3.2 Software Developer

The software developer is a proficient Java developer who has been tasked with implementing some cryptographic protocol either because nobody in the team has any experience or because they are working on their own. Having attended a number of Computer Security lectures as an undergraduate they know enough to know that they do not know enough to attempt to create their own protocol. Here they have two options, ideally they find a LySa model of a suitable protocol which they can run through Hajyle to generate their skeleton Java code. More likely they are forced to look through

literature to find a protocol they believe is suitable. Once they have identified a potential protocol they are then tasked to try to find enough material to enable them to implement the protocol, perhaps a basic protocol narration along with a text-based explanation which fills in the gaps in knowledge. For either of these options they perform a cursory check to see if any attacks have been discovered.

Naturally the implementation process has a few false starts and misunderstandings but eventually the developer gets a working implementation and uses Elyjah to generate the LySa representation. They can then use the LySatool to check the security properties of the protocol. Perhaps the LySatool reports some errors at this point. The developer then checks whether this is a problem that the original protocol suffers from by undertaking more focused research. Otherwise they can use the AnaLySa and VisuaLySa tools to check the protocol narration is the same as expected from the original specification and that they have implemented the protocol successfully.

If changes are needed to be made to the protocol this can either be done on the LySa model and converted to Java via Hajyle or on the original Java depending on the developer's confidence. The advantage with the second option is that the file retains the extra code needed to connect with the rest of their application which would have to be reimplemented if using Hajyle. Under either option though the implementation can be rechecked at any time using Elyjah and the LySatool. Once a secure protocol has been implemented, the necessary liveness checks can be made by running the implementation and examining the output.

It is likely the developer knows the deployment scenario the protocol is intended for. They can then use Elyjah to generate Meta-LySa for this deployment scenario and validate that this does not introduce other errors. Additionally PAMela can be run on the Elyjah-generated LySa and results from actual implementation runs on the target devices used as rates to give the developer the most accurate PAM model possible. From this they can choose appropriate encryption and communication algorithms or network infrastructure improvements to best optimise the protocol's run-time performance. Once they have settled on these details, service-level agreements can be provided using the probability and cumulative density graphs provided by PAMeLa.

8.4 Conclusions

8.4.1 LyTE

We developed a series of tools to assist users unfamiliar with process calculi utilise the security analysis of the LySatool. With the LySa Editor's parse checker and ability to check for matching send/receive processes we have made it easier to write or edit LySa models and with the AnaLySa and VisuaLySa we provide tools to translate a LySa process into a simpler model. Integration with the LySatool means users can observe any potential security breaches in a number of intuitive ways. Attacker's actions are visualised and non-confidential message exchanges are highlighted. As a side-effect of this work we have provided a user of the Elyjah tool with the necessary tools to check that the LySa output of Elyjah is accurate with respect to the Java implementation. We believe this is more useful to the users of Elyjah than a theoretical proof of the correctness of Elyjah which may be hard to understand. Additionally a simple case-by-case proof is more relevant to the user.

With Elyjah, we tried to provide the functionality of cryptographic model analysis to a user who had no inclination to learn how to use these formal models. With the LySa Toolkit in Eclipse the intention was to try to create some tools which appeal to those who wish to take the next step and have more understanding of the output of Elyjah and the underlying LySa.

These tools have been used for the last two years in the MSc Language-based Security course at DTU where the students were actively encouraged to use LyTE. Feedback from course organisers was very positive as they noted "It definitely improved students understanding of how a protocol is modelled with LySa process calculus, and how the tool implementing the analysis works"[80]. It was reported that students saved time both in debugging their models and interpreting the analysis results and the plug-in was declared a "great help"[80] for LySa users.

8.4.2 PAMeLA

It is important for implementers to be able to make sure that they choose an appropriate protocol for their deployment scenario. For protocol developers it is equally important that their design is not so flawed as to include unnecessary performance bottlenecks. PAMeLa allows protocol developers to utilise powerful performance analysis without needing any knowledge of the underlying mechanics or language. With the aid of the

AutoRate plug-in appropriate rates are automatically calculated based on a user's selected deployment scenario, eliminating a difficult aspect of performance modelling. With these tools, network administrators can work out the best way of improving the performance of their network thus potentially saving time and money instead of accidentally devoting resources to areas which would not improve performance.

With the passage-time analysis available, users can provide service-level agreements giving reasonable estimates as to when a protocol will terminate. Combined with then generating implementations using Hajyle and using rates by using final implementations of JaLAPI they will also then be able to use these service-level agreements to potentially spot message exchanges which fall outside of expected range which may indicate an attempted attack.

8.4.3 Elyjah and Hajyle

Elyjah translates a Java implementation of a cryptographic protocol into a formal model. Such a process can only be reliably achieved using automation. The LySa model can be analysed using the LySatool to return the security properties of the protocol implementation. The examples given in Chapters 5 and 6 are necessarily brief so that enough detail could be given to the explanation. In practice, Elyjah and Hajyle have been used to translate protocols such as the Yahalom[22], Otway-Rees[62] and Needham-Schroeder[53] protocols. These implementations have then been used as part of small applications such as secure instant-messaging. The intent is to allow these implementations to be used on Java2 ME or Android devices such as mobile phones where there is a market for secure communication and authentication.

Several library classes are provided to allow a developer to develop a protocol. Elyjah has been designed so that it can be used in a real program by replacing the implementation of the `Network` class. While Elyjah cannot necessarily analyse the source code of an arbitrary communications package, it is possible to implement a protocol that can be translated by Elyjah and used in a full distributed application.

Hajyle allows those developers already familiar with LySa to generate implementations based on LySa models. This tool also allows continued development once a user has generated a LySa model using Elyjah. Using tools such as those detailed in Chapter 3 helps a user to work with the generated LySa model and create an updated Java implementation once any changes have been made.

8.4.4 General Conclusions

In Chapter 1 we defined verification and validation of programs as making the program right and making the right program respectively. By converting Java implementations into LySa models we can verify the security properties of the protocol. Using the tools in the LySa Toolkit the user is presented with clear visualisations of the protocol to validate that the protocol they implemented will function as they intended. Additionally the performance analysis provided through PAMeLa provides additional quality assurance with respect to system usability that is integral to making the right program.

Over the course of developing these tools it has been made clear that excellent formal methods are available for analysing security in applications that are not being used by developers at large. This is partly due to the shortage of available literature regarding these methods targeted towards complete novices in process calculi. Additionally while many universities offer courses on these topics they are optional so a graduate may not have been exposed to them. With little information available at present for ordinary developers, further development of the sort of tools developed here could really help to introduce more people to these formal methods.

A quote attributed to Tom Melham states that “Formal methods will never have a significant impact until they can be used by people that don’t understand them”. Tools like the ones presented in this dissertation are examples of how we can make real progress towards this goal.

Bibliography

[1] Martín Abadi and Andrew D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[2] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalised Stochastic Petri Nets*. John Wiley and Sons, 1995.

[3] Ross J. Anderson and Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, January 2001.

[4] David Aspinall, Stephen Gilmore, Martin Hofmann, Donald Sannella, and Ian Stark. Mobile Resource Guarantees for Smart Devices. In *Proceedings of CASSIS04, LNCS*, pages 1–26. Springer, 2005.

[5] Michael Backes, Catalin Hritcu, and Matteo Maffei. Automated Verification of Remote Electronic Voting Protocols in the Applied Pi-Calculus. In *CSF '08: Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, pages 195–209. IEEE Computer Society, 2008.

[6] Michael J. Beller, Li-Fung Chang, and Yacov Yacobi. Privacy and Authentication on a Portable Communications System. *IEEE Journal on Selected Areas in Communications*, 11(6):821–829, 1993.

[7] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement Types for Secure Implementations. In *Computer Security Foundations Symposium*, pages 17–32, 2008.

[8] Marco Bernardo, Lorenzo Donatiello, and Roberto Gorrieri. A formal approach to the integration of performance aspects in the modeling and analysis of concurrent systems. *Information and Computation*, 144(2):83 – 154, 1998.

[9] Karthikeyan Bhargavan, Cedric Fournet, Andrew D. Gordon, and Stephen Tse. Verified Interoperable Implementations of Security Protocols. *Proceedings of the Computer Security Foundations Workshop*, 2006:139 – 152, 2006.

[10] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[11] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Nielson. Automatic validation of protocol narration. *Proceedings of 16th IEEE Computer Security Foundations Workshop (CSFW 16)*, pages 126–140, 2003.

[12] C. Bodei, M. Buchholtz, P. Degano, H. Riis Nielson, and F. Nielson. Static Validation of Security Protocols. *Journal of Computer Security*, 13(3):347–390, 2005.

[13] Chiara Bodei, Linda Brodo, Pierpaolo Degano, and Han Gao. Detecting and Preventing Type Flaws at Static Time. *J. Comput. Secur.*, 18(2):229–264, 2010.

[14] Chiara Bodei, Mikael Buchholtz, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Control Flow Analysis Can Find New Flaws Too. *Proceedings of Workshop on Issues in the Theory of Security (WITS 04)*, 2004.

[15] Chiara Bodei, Michele Curti, Pierpaolo Degano, Mikael Buchholtz, Flemming Nielson, Hanne Riis Nielson, and Corrado Priami. Performance Evaluation of Security Protocols Specified in LySa. *Electr. Notes Theor. Comput. Sci.*, 112:167–189, 2005.

[16] G. Bolch, S. Greiner, H. de Meer, and K. Trivedi. *Queueing Networks and Markov Chains*. Wiley, 2006.

[17] Jeremy T. Bradley, Nicholas J. Dingle, Stephen T. Gilmore, and William J. Knottenbelt. Extracting Passage Times from PEPA models with the HYDRA Tool: A Case Study. In *UKPEW 2003, 19th UK Performance Engineering Workshop*, pages 79–90, June 2003.

[18] Sébastien Briais and Uwe Nestmann. A formal semantics for protocol narrations. *Theor. Comput. Sci.*, 389(3):484–511, 2007.

[19] M. Buchholtz. User's Guide for the LySatool version 2.01. April 2005.

[20] M. Buchholtz. Implementing Control Flow Analysis for Security Protocols. *DEGAS Report WP6-IMM-I00-Pub-003*, Draft 2003.

[21] Mikael Buchholtz. *Automated Analysis of Networking Systems*. PhD thesis, Technical University of Denmark, 2004.

[22] Michael Burrows, Martín Abadi, and Roger Needham. A Logic of Authentication. In *Proceedings of the Royal Society*, volume 426. 1989.

[23] Alex Busenius. Expi2Java An Extensible Code Generator for Security Protocols. Master's thesis, Saarland University, 2008.

[24] Carlos Caleiro, Luca Viganò, and David Basin. Deconstructing Alice and Bob. *Electron. Notes Theor. Comput. Sci.*, 135(1):3–22, 2005.

[25] G. Cattaneo, A. De Santis, and U. Ferraro Petrillo. Visualization of cryptographic protocols with GRACE. *J. Vis. Lang. Comput.*, 19(2):258–290, 2008.

[26] Sagar Chaki and Anupam Datta. ASPIER: An Automated Framework for Verifying Security Protocol Implementations. In *CSF '09: Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, pages 172–185, Washington, DC, USA, 2009. IEEE Computer Society.

[27] Harshal S. Chhaya and Sanjay Gupta. Performance modeling of asynchronous data transfer methods of IEEE 802.11 MAC protocol. *Wirel. Netw.*, 3(3):217–234, 1997.

[28] Xavier Didelot. COSP-J: A Compiler for Security Protocols. Master's thesis, Oxford University Computing Laboratory, 2003.

[29] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[30] D. Dolev and A.C. Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, IT-29(2):198 – 208, 1983/03/.

[31] Eclipse IDE. http://www.eclipse.org/, 2007. Webpage hosted by the Eclipse Foundation.

[32] Niklas Elmqvist. Protoviz - A Simple Protocol Visualization, April 2004. www.math.chalmers.se/ elm/courses/security/.

[33] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM.

[34] Han Gao. *Analysis of Security Protocols by Annotations*. PhD thesis, Technical University of Denmark (DTU), 2008.

[35] Han Gao, Chiara Bodei, Pierpaolo Degano, and Hanne Riis Nielson. A Formal Analysis for Capturing Replay Attacks in Cryptographic Protocols. In *ASIAN'07: Proceedings of the 12th Asian computing science conference on Advances in computer science*, pages 150–165, Berlin, Heidelberg, 2007. Springer-Verlag.

[36] J. Goubault-Larrecq and F. Parrennest. Cryptographic Protocol Analysis on Real C Code. *Verification, Model Checking, and Abstract Interpretation. 6th International Conference, VMCAI 2005. Proceedings*, pages 363 – 79, 2005//.

[37] Jean Goubault-Larrecq. Towards Producing Formally Checkable Security Proofs, Automatically. pages 224–238. IEEE Computer Society, 2008.

[38] Holger Hermanns and Michael Rettelbach. Syntax, Semantics, Equivalences, and Axioms for MTIPP. In *in Proc. of the 2nd Workshop on Process Algebras and Performance Modelling (PAPM '94*, pages 71–87, 1994.

[39] Jane Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, New York, NY, USA, 1996.

[40] Tony Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[41] Chul-Wuk Jeon, Il-Gon Kim, and Jin-Young Choi. Automatic Generation of the C# Code for Security Protocols Verified with Casper/FDR. In *AINA '05: Proceedings of the 19th International Conference on Advanced Information Networking and Applications*, pages 507–510, Washington, DC, USA, 2005. IEEE Computer Society.

[42] S. C. Johnson. Lint, a C Program Checker. In *COMP. SCI. TECH. REP*, pages 78–1273, 1978.

[43] Jan Jurjens. Security Analysis of Crypto-based Java Programs using Automated Theorem Provers. In *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 167–176, Washington, DC, USA, 2006. IEEE Computer Society.

[44] Jan Jurjens and Mark Yampolskiy. Code Security Analysis with Assertions. In *ASE '05: Proceedings of the 20th IEEE International Conference on Automated Software Engineering (ASE'05)*, pages 392–395, 2005.

[45] Gavin Lowe. An Attack on the Needham-Schroeder Public-Key Authentication Protocol. *Information Processing Letters*, 56(3):131–133, 1995.

[46] Gavin Lowe. Casper: A Compiler for the Analysis of Security Protocols. In *Journal of Computer Security*, pages 53–84. Society Press, 1998.

[47] Simon Lukell, Christopher Veldman, and Andrew Hutchison. Automated Attack Analysis and Code Generation in a Multi-Dimensional Security Protocol Engineering Framework. In *In Southern African Telecommunication Networks and Applications Conference (SATNAC)*, 2003.

[48] Jonathan Millen and Frederic Muller. Cryptographic Protocol Generation From CAPSL. Technical Report SRI-CSL-01-07, SRI International, December 2001.

[49] Jonathan K. Millen and Vitaly Shmatikov. Constraint Solving for Bounded-Process Cryptographic Protocol Analysis. In *ACM Conference on Computer and Communications Security*, pages 166–175, 2001.

[50] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.

[51] National Institute of Standards and Technology. Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family. Technical report, DEPARTMENT OF COMMERCE, November 2007.

[52] George C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM.

[53] Roger M. Needham and Michael D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Commun. ACM*, 21(12):993–999, 1978.

[54] F. Nielson and H. Seidl. Succinct Solvers. Technical Report 01–12, Universität Trier, 2001.

[55] F. Nielson, H. Seidl, and H. Nielson. A Succinct Solver for ALFP. *Nordic Journal of Computing*, 9:335–372, 2002.

[56] Hanne Riis Nielson and Flemming Nielson. Flow Logic: A Multi-paradigmatic Approach to Static Analysis. *The Essence of Computation: Complexity, Analysis, Transformation*, pages 223–244, 2002.

[57] oCERT 2008-016 Multiple OpenSSL Signature Verification API Misuse. http://ocert.org/advisories/ocert-2008-016.html.

[58] Nicholas O'Shea. Protocol Analysis in a new LyTE. In *Proceedings of The 13th Nordic Workshop on Secure IT Systems*, pages 83–94, 2008.

[59] Nicholas O'Shea. Using Elyjah to Analyse Java Implementations of Cryptographic Protocols. In *FCS-ARSPA-WITS'08*, pages 211–226, 2008.

[60] Nicholas O'Shea. Concerning Performance Driven Cryptographic Protocol Development. In *8th Workshop on Process Algebra and Stochastically Timed Activities*, 2009.

[61] Dave Otway and Owen Rees. Efficient and Timely Mutual Authentication. *SIGOPS Oper. Syst. Rev.*, 21(1):8–10, 1987.

[62] Dave Otway and Owen Rees. *Efficient and timely mutual authentication*, volume 21. ACM, New York, NY, USA, 1987.

[63] Lawrence C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6:85–128, 1998.

[64] Alfredo Pironti. *Sound Automatic Implementation Generation and Monitoring of Security Protocol Implementations from Verified Formal Specifications*. PhD thesis, Politecnico di Torino (Italy), 2010.

[65] A. Pnueli, M. Siegel, and E. Singerman. Translation Validation. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 151–166. Springer, 1998.

[66] Erik Poll and Aleksy Schubert. Verifying an implementation of SSH. *WITS*, pages 164–177, 2007.

[67] Davide Pozza, Riccardo Sisto, and Luca Durante. Spi2Java: Automatic Cryptographic Protocol Java Code Generation from spi Calculus. In *AINA '04: Proceedings of the 18th International Conference on Advanced Information Networking and Applications*, page 400, Washington, DC, USA, 2004. IEEE Computer Society.

[68] A. Regev, W. Silverman, and E. Shapiro. Representation and Simulation of Biochemical Processes Using the pi-Calculus Process Algebra. In *Pacific Symposium on Biocomputing*, pages 459–470, 2001.

[69] A. W. Roscoe. Model-checking CSP. *A Classical Mind: Essays in Honour of C. A. R. Hoare*, pages 353–378, 1994.

[70] Elton Saul and Andrew Hutchison. SPEAR II - The Security Protocol Engineering and Analysis Resource. *Proc. 2nd Annual South African Telecommunications, Networks and Applications Conference*, pages 171 – 177, 1999.

[71] Bruce Schneier. *Secrets and Lies: Digital Security in a Networked World*. Wiley, 1 edition, January 2004.

[72] Dino Schweitzer, Leemon Baird, Michael Collins, Wayne Brown, and Michael Sherman. GRASP: A Visualization Tool for Teaching Security Protocols. In *Proceedings from the 10th Colloquium for Information Systems Security Education*, 2006.

[73] Dawn Xiaodong Song, Sergey Berezin, and Adrian Perrig. Athena: A Novel Approach to Efficient Automatic Security Protocol Analysis. *Journal of Computer Security*, 9(1/2):47–74, 2001.

[74] Dawn Xiaodong Song, Adrian Perrig, and Doantam Phan. AGVI - Automatic Generation, Verification, and Implementation of Security Protocols. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 241–245, London, UK, 2001. Springer-Verlag.

[75] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, 1977.

[76] Thorsten Tarrach. Spi2F# A Prototype Code Generator for Security Protocols. Master's thesis, Saarland University, 2008.

[77] Benjamin Tobler and Andrew C. M. Hutchison. Generating Network Security Protocol Implementations from Formal Specifications. In *in Proc. IFIP World Comp. Congress Workshop on Certification and Security in Inter-Organizational E-Services (CSES*, 2004.

[78] M. Tribastone, A. Duguid, and S. Gilmore. The PEPA Eclipse Plug-in. *Performance Evaluation Review*, 36(4):28–33, March 2009.

[79] Xiaoyun Wang, Yiqun L. Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. *Lecture Notes in Computer Science*, 3621:17–36, November 2005.

[80] Ender Yuksel. LyTE Feedback. Personal Communication, October 2010.

[81] Jelena Zaitseva. TECP Tutorial Environment for Cryptographic Protocols. Master's thesis, Institute of Computer Science, University of Tartu, 2003.